

# PYTHON

## Histoire urbaine digitale : Lausanne Time Machine

*Enseignement SHS Master*

EPFL | 2021-22

---

# TABLE DES MATIÈRES

|   |    |
|---|----|
| <b>Structure</b>                                | 1  |
| <b>Licence</b>                                  | 2  |
| <b>Préface</b>                                  | 2  |
| <b>Installation</b>                             | 3  |
| 1. <b>Premiers pas</b>                          | 4  |
| 2. <b>Flux d'exécution</b>                      | 9  |
| 3. <b>Instructions répétitives</b>              | 14 |
| 4. <b>Principaux types de données</b>           | 18 |
| 5. <b>Fonctions prédéfinies</b>                 | 25 |
| 6. <b>Fonctions originales</b>                  | 28 |
| 7. <b>Manipuler des fichiers</b>                | 35 |
| 8. <b>Approfondir les structures de données</b> | 40 |
| 9. <b>Cartographier et visualiser</b>           | 47 |
| 10. <b>Fouiller et étudier les textes</b>       | 53 |

---

# STRUCTURE

|               |  | Chapitres |   |   |   |   |   |   |   |   |    |
|---------------|--|-----------|---|---|---|---|---|---|---|---|----|
|               |  | 1         | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Débutant      |  |           |   |   |   |   |   |   |   |   |    |
|               |  |           |   |   | * |   |   |   |   |   |    |
| Intermédiaire |  |           |   |   |   |   | * |   |   |   |    |
| Avancé        |  |           |   |   |   |   |   |   |   | * |    |

\* = validation des connaissances

Ce polycopié est structuré comme un cours continu de 10 chapitres. La difficulté augmente graduellement, de sorte qu'il est possible de commencer à le lire sans aucune base de programmation. Les six premiers chapitres abordent les notions élémentaires de la programmation en Python : types de données et structures de données élémentaires, opérations et fonctions. Ils permettent aux débutant-e-s de s'initier à la pensée

computationnelle par la pratique. Les « faux-débutant-e-s » et les élèves de niveau intermédiaire, ayant déjà des notions de programmation dans un langage autre que Python pourront y acquérir rapidement les bases de ce langage. Les chapitres 7 et 8 offrent les outils nécessaires pour travailler avec les principaux types de données en histoire urbaine digitale, tandis que les deux derniers chapitres abordent des concepts plus avancés, comme les interfaces géographiques, et le *distant reading*. Ces quatre derniers chapitres permettront aux programmeur-euse-s ayant intégré les notions de base de Python d'acquérir des compétences plus spécifiques pour la manipulation de données textuelles ou géographiques en humanités digitales.

---

## LICENCE

Ce polycopié est une adaptation de l'ouvrage *Apprendre à programmer avec Python 3* de Gérard Swinnen. Voir [inforef.be/swi/python.htm](http://inforef.be/swi/python.htm)

L'ouvrage qui suit est distribué suivant les termes de la Licence **Creative Commons** « Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique - 2.0 France ».

Cela signifie que vous pouvez copier, modifier et redistribuer ces pages tout à fait librement, pour autant que vous respectiez un certain nombre de règles qui sont précisées dans cette licence, dont le texte complet peut être consulté dans l'annexe C, page [445](#) de l'ouvrage cité ci-dessus.

Pour l'essentiel, sachez que vous ne pouvez pas vous approprier ce texte pour le redistribuer ensuite (modifié ou non) en définissant vous-même d'autres droits de copie. Le document que vous redistribuez, modifié ou non, doit obligatoirement inclure intégralement le texte de la licence citée ci-dessus, le présent avis et la préface qui suit. L'accès à ces notes doit rester libre pour tout le monde. Vous êtes autorisé à demander une contribution financière à ceux à qui vous redistribuez ces notes, mais la somme demandée ne peut concerner que les frais de reproduction. Vous ne pouvez pas redistribuer ces notes en exigeant pour vous-même des droits d'auteur, ni limiter les droits de reproduction des copies que vous distribuez. La diffusion commerciale de ce texte en librairie, sous la forme classique d'un manuel imprimé, est réservée exclusivement à la maison d'édition Eyrolles (Paris).

---

## PRÉFACE

La préface de l'auteur original est disponible intégralement sur ce lien : [inforef.be/swi/python.htm](http://inforef.be/swi/python.htm).

---

# INSTALLATION

## MacOS 10.13 et ultérieur

1. Téléchargez Anaconda sur le lien suivant :  
<https://www.anaconda.com/products/individual#macos>
2. Double-cliquez sur le paquet téléchargé et suivez la procédure d'installation.
3. Il n'est pas nécessaire d'installer PyCharm.

## MacOS 10.10-10.12

Téléchargez Anaconda 2019 : [https://repo.anaconda.com/archive/Anaconda3-2019.10-MacOSX-x86\\_64.pkg](https://repo.anaconda.com/archive/Anaconda3-2019.10-MacOSX-x86_64.pkg), puis procédez comme ci-dessus.

## Windows 8 et ultérieur

1. Téléchargez Anaconda sur le lien suivant :  
<https://www.anaconda.com/products/individual#windows>
2. Double-cliquez sur le paquet téléchargé et suivez la procédure d'installation.  
Lorsque l'installateur vous le demande, n'ajoutez pas Anaconda à la variable d'environnement PATH, mais enregistrez Anaconda comme votre programme par défaut pour Python.
3. Il n'est pas nécessaire d'installer PyCharm.

## Windows 7

Téléchargez Anaconda 2019 64 bits [https://repo.anaconda.com/archive/Anaconda3-2019.10-Windows-x86\\_64.exe](https://repo.anaconda.com/archive/Anaconda3-2019.10-Windows-x86_64.exe) (ou <https://repo.anaconda.com/archive/Anaconda3-2019.10-Windows-x86.exe>, si votre ordinateur ne prend pas en charge les programmes 64 bits), puis procédez comme ci-dessus.

## Linux

Suivez la procédure d'installation décrite à l'adresse suivante :  
<https://docs.anaconda.com/anaconda/install/linux/>

## Autres

Référez-vous à la documentation officielle :  
<https://docs.anaconda.com/anaconda/install/index.html>

---

# 1. PREMIERS PAS

*La programmation est l'art de commander à un ordinateur de faire exactement ce que vous voulez, et Python compte parmi les langages qu'il est capable de comprendre pour recevoir vos ordres. Nous allons essayer cela tout de suite avec des ordres très simples concernant des nombres, puisque les nombres constituent son matériau de prédilection. Nous allons lui fournir nos premières « instructions », et préciser au passage la définition de quelques termes essentiels du vocabulaire informatique, que vous rencontrerez constamment dans la suite de cet ouvrage.*

## Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Grâce à Jupyter notebook, vous allez l'utiliser en mode interactif, c'est-à-dire de manière à dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage.

Vous pouvez tout de suite utiliser l'ordinateur comme une simple calculatrice de bureau. Veuillez donc vous-même tester les commandes ci-dessous (créez une nouvelle cellule pour chaque commande).

```
5+3  
2 - 9  
7+3 *2  
( 7+3 )*2  
20/3  
20//3
```

Les espaces sont optionnels.

Question: La hiérarchie des opérations mathématiques est-elle respectée ?

Veuillez remarquer au passage une règle qui vaut dans tous les langages de programmation : les conventions mathématiques de base sont celles qui sont en vigueur dans les pays anglophones. Le séparateur décimal y est donc toujours un point, et non une virgule comme chez nous. Notez aussi qu'en informatique, les nombres réels sont souvent désignés comme des nombres « à virgule flottante » (*floating point numbers*, ou simplement *float*).

L'**opérateur** de division / effectue une division réelle. Si vous souhaitez obtenir une division entière (c'est-à-dire dont le résultat – tronqué – ne peut être qu'un entier, vous devez utiliser l'opérateur //). L'opérateur %, appelé **modulo**, permet d'obtenir le reste de la division entière.

## Données et variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses (tout ce qui est numérisable, en fait), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite de nombres binaires.

Pour distinguer contenus divers les uns des autres, le langage de programmation fait usage de différents **types** de **variables** (le type *int* (*entier*), le type *float*, le type *string* (*chaîne de caractères*), le type *list* (*liste*), etc.). Nous allons expliquer tout cela dans les pages suivantes.

### Noms de variables et mots réservés

Les **noms de variables** sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que **altitude**, **altit** ou **alt** conviennent mieux que **x** ou **valeur** pour exprimer une altitude.

En Python, les noms de variables doivent en outre obéir à quelques règles simples :

- Un nom de variable est une séquence de lettres (a → z , A → Z) et de chiffres (0 → 9), qui doit toujours commencer par une lettre.
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc. sont interdits, à l'exception du caractère \_ (souligné).
- La casse est significative (les caractères majuscules et minuscules sont distingués).

*Attention : Joseph, joseph, JOSEPH sont donc des variables différentes.*

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

|      |          |        |        |       |          |        |
|------|----------|--------|--------|-------|----------|--------|
| and  | as       | assert | break  | class | continue | def    |
| del  | elif     | else   | except | False | finally  | for    |
| from | global   | if     | import | in    | is       | lambda |
| None | nonlocal | not    | or     | pass  | raise    | return |
| True | try      | while  | with   | yield |          |        |

## Affectation (ou assignation)

Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons **définir une variable** et lui **affecter une valeur**. « Affecter une valeur » signifie qu'on établit un lien entre le nom de la variable et sa valeur (son contenu).

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe *égale* :

```
n = 7  
prenom_nom = "Samuel Secretan"  
age = 9.5
```

Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

- trois noms de variables, à savoir **n**, **nom\_famille** et **age** ;
- trois séquences d'octets, où sont encodées le nombre entier **7**, la chaîne de caractères **Samuel Secretan** et le nombre réel **9.5**.

## Afficher la valeur d'une variable

À la suite de l'exercice ci-dessus, nous disposons donc des trois variables **n**, **prenom\_nom** et **age**.

Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à écrire à la dernière ligne d'une cellule le nom de la variable, puis à cliquer sur *Run*. Python répond en affichant la valeur correspondante. À l'intérieur d'un programme, vous utiliserez cependant toujours la fonction **print()**. Testez maintenant ces deux options.

```
prenom_nom
```

```
print(prenom_nom)
```

Remarquez la subtile différence dans les affichages obtenus avec chacune des deux méthodes. La fonction **print()** n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée, alors que l'autre méthode (celle qui consiste à entrer seulement le nom de la variable) affiche aussi des apostrophes afin de vous rappeler que la variable traitée est du type « chaîne de caractères » (nous y reviendrons).

## Type des variables

En Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'affecter une valeur à un nom de variable pour que celle-ci soit *automatiquement créée avec le type qui correspond au mieux à la valeur fournie*. Dans l'exercice précédent, par exemple, les variables **n**, **prenom\_nom** et **age** ont été créées automatiquement chacune avec un type différent (« nombre entier » pour **n**, « chaîne de caractères » pour **prenom\_nom**, « nombre à virgule flottante » (ou *float*) pour **age**).

Vous pouvez à tout moment vérifier le type d'une variable en utilisant la fonction **type()**.

```
type(n)
type(prenom_nom)
type(age)
```

## Priorité des opérations

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de *règles de priorité*. En Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées en mathématiques. Vous pouvez les mémoriser aisément à l'aide d'un « truc » mnémotechnique, l'acronyme *PEMDAS* : Parenthèses, Exposants, Multiplication/Division, Addition/Soustraction.

## Composition

L'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
print(17+3)
```

Attention, cependant, il y a une limite à ce que vous pouvez combiner ainsi. Dans une expression, ce que vous placez à la gauche du signe égale doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, il s'agit d'un symbole *d'affectation* (nous plaçons un certain contenu dans une variable) et non d'un symbole d'égalité. Le symbole d'égalité (dans un test conditionnel, par exemple) sera évoqué un peu plus loin.

Ainsi par exemple, l'instruction **m + 1 = b** est illégale.

Par contre, écrire **a = a + 1** est inacceptable en mathématique, alors que cette forme d'écriture est très fréquente en programmation. L'instruction **a = a + 1** signifie en l'occurrence « augmenter la valeur de la variable a d'une unité » (ou encore : « incrémenter a »).

## **Exercices**

- 1 Décrivez le plus clairement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

```
longueur_parcelle = 20  
largeur_parcelle = 5*9.3  
print(longueur_parcelle * largeur_parcelle)
```

- 2 Testez la commande ci-dessous. Imprimez ensuite les variables **longueur** et **largeur**. Que constatez-vous ?

```
longueur = largeur = 10
```

- 3 Testez la commande ci-dessous. Imprimez ensuite les variables **longueur**, **largeur** et **hauteur**. Que constatez-vous ?

```
longueur, largeur, hauteur = 15, 25, 6
```

---

## 2. FLUX D'EXÉCUTION

*L'activité essentielle d'un-e programmeur-euse est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un **algorithme**.*

*Le « chemin » suivi par Python à travers un programme est appelé un **flux d'exécution**, et les constructions qui le modifient sont appelées des **instructions de contrôle de flux**.*

*Les **structures de contrôle** sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois : la **séquence** et la **sélection**, que nous allons décrire dans ce chapitre, et la **répétition** que nous aborderons au chapitre suivant.*

### Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, *dans l'ordre où elles ont été écrites à l'intérieur du script*.

Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
a = 3
b = 7
a = b
b = a
print(a, b)
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2<sup>e</sup> et 3<sup>e</sup> lignes. N'hésitez pas à le vérifier par vous-même pour bien comprendre cet exemple.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une **instruction conditionnelle** comme l'instruction **if** décrite ci-après (nous en rencontrerons d'autres plus loin, notamment à propos des boucles de répétition). Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

## Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des algorithmes véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de *tester une certaine condition* et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction **if**. Pour expérimenter son fonctionnement, veuillez entrer dans votre éditeur Python les deux lignes suivantes :

```
year = 1845
if (year > 1830):
    print("Né après la révolution libérale vaudoise")
```

La première commande affecte la valeur 1845 à la variable **year**. Jusqu'ici rien de nouveau. Notez que la dernière ligne d'instruction est précédée d'une tabulation. En Python, cette **indentation** est *obligatoire* et l'absence de tabulation entraînera une erreur, comme vous pouvez le vérifier facilement. La tabulation permet également de mieux structurer votre code.

Recommencez le même exercice, mais avec **year = 1820** en guise de première ligne : cette fois Python n'affiche plus rien.

L'expression que vous avez placée entre parenthèses après **if** est ce que nous appellerons désormais une **condition**. L'instruction **if** permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons indentée après le double point (**:**) est exécutée. Si la condition est fausse, rien ne se passe. Notez que les parenthèses utilisées ici avec l'instruction **if** sont optionnelles : nous les avons utilisées pour améliorer la lisibilité.

Recommencez encore, en ajoutant deux lignes supplémentaires à la suite des précédentes, dans la même cellule. Veillez bien à ce que la quatrième ligne débute tout à fait à gauche (pas d'indentation), mais que la cinquième soit à nouveau indentée (de préférence avec un retrait identique à celui de la troisième) :

```
else:
    print("Né avant la révolution libérale vaudoise")
```

Comme vous l'aurez certainement déjà compris, l'instruction **else** permet de programmer une exécution *alternative*, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « *else if* ») :

```

year = 1845
if (year > 1830):
    print("Né après la révolution libérale vaudoise")
elif (year < 1830):
    print("Né avant la révolution libérale vaudoise")
else:
    print("Né l'année de la révolution libérale vaudoise")

```

## Opérateurs de comparaison

La condition évaluée après l'instruction **if** peut contenir les opérateurs de **comparaison** suivants :

```

x == y      # x est égal à y
x != y      # x est différent de y
x > y       # x est plus grand que y
x < y       # x est plus petit que y
x >= y      # x est plus grand ou égal à y
x <= y      # x est plus petit ou égal à y

```

## Instructions composées – blocs d'instructions

La construction que vous avez utilisée avec l'instruction **if** est votre premier exemple d'**instruction composée**. Vous en rencontrerez bientôt d'autres. En Python, les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête.

Exemple :

```

Début du code
Ligne d'en-tête :
    Première instruction du bloc
    ...
    Dernière instruction du bloc
Suite du code

```

S'il y a plusieurs instructions sous la ligne d'en-tête, *elles doivent toutes être indentées de manière identique*. Ces instructions indentées constituent ce que nous appellerons désormais un **bloc d'instructions**. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête.

## Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes. Exemple :

```
if canton_bourgeoisie == "Vaud":                      # 1
    if masculin == True:                            # 2
        if duree_residence >= 1:                  # 3
            if age >= 30:                         # 4
                print("Est éligible et électeur")   # 5
            elif age >= 23:                         # 6
                print("Est électeur")                 # 7
            else:                                # 8
                print("N'est pas résident depuis assez longtemps")
        print("Est un citoyen")                     # 10
```

Analysez cet exemple. Ce fragment de programme n'imprime la phrase « *Est éligible et électeur* » que dans le cas où les quatre premières conditions testées sont vraies. Si la variable **age** n'est pas supérieure à 30 ans, mais que les trois premières conditions sont vraies, le compilateur teste la condition de la ligne 6 : **elif age >= 23:**. Dans le cas où celle-ci est vraie, la phrase « *Est électeur* » s'affiche.

Pour que la phrase « *Est un citoyen* » soit affichée, il faut et il suffit que les deux premières conditions soient vraies. L'instruction d'affichage de cette phrase (ligne 10) se trouve en effet au même niveau d'indentation que l'instruction : **if duree\_residence >= 1:** (ligne 3). Les deux font donc partie d'un même bloc, lequel est entièrement exécuté si les conditions testées aux lignes 1 et 2 sont vraies.

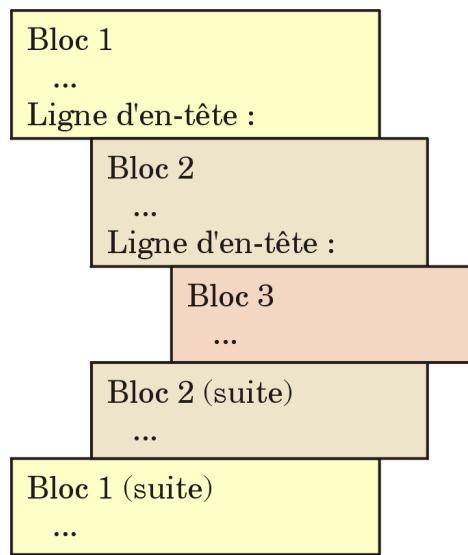
Si les deux premières conditions sont vérifiées, mais que la troisième est fausse, alors la phrase « *N'est pas résident depuis assez longtemps* » s'affiche dans tous les cas.

## Quelques règles de syntaxe Python

Les limites des instructions et des blocs sont définies par la mise en page. Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). En Python, c'est le caractère de fin de ligne qui joue ce rôle. On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial **#**. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Dans la plupart des autres langages, un bloc d'instructions doit être délimité par des symboles spécifiques (parfois même par des instructions, telles que **begin** et **end**). En Javascript, par exemple, un bloc d'instructions doit être délimité par des accolades. Cela permet d'écrire les blocs d'instructions les uns à la suite des autres, sans se préoccuper d'indentation ni de sauts à la ligne, mais cela peut conduire à l'écriture de programmes confus et difficiles à relire. Avec Python, vous devez utiliser les sauts à la ligne et

l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En prime, vous obtenez un code plus propre et lisible.



*Les blocs d'instruction sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (*if*, *elif*, *else*, *while*, etc.) se terminant pas un double point. Ils sont délimités par un niveau d'indentation identique.*

### Exercices

- 1 Définissez une variable nommée **annee\_naissance**, puis rédigez une instruction conditionnelle qui permette de vérifier que le personnage est né entre 1732 et 1818. Si oui, le programme devra afficher : "Peut être chef-fe de famille". Sinon, le programme devra afficher : "Ne peut pas être chef-fe de famille".
- 2 Ajoutez une seconde variable nommée **annee\_actuelle** et modifiez le code de l'exercice précédent pour que l'algorithme vérifie automatiquement que l'âge du personnage est compris entre 14 et 100 ans, quelle que soit la valeur de **annee\_actuelle**.

### 3. INSTRUCTIONS RÉPÉTITIVES

*L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction while.*

#### Réaffectation

Nous ne l'avions pas encore signalé explicitement : il est permis de réaffecter une nouvelle valeur à une même variable, autant de fois qu'on le souhaite. L'effet d'une réaffectation est de remplacer l'ancienne valeur d'une variable par une nouvelle.

```
hauteur_du_lac = 375
print(hauteur_du_lac)      # 375 s'affiche
hauteur_du_lac = 372
print(hauteur_du_lac)      # 372 s'affiche
```

Ceci nous amène à attirer une nouvelle fois votre attention sur le fait que le symbole égale utilisé en Python pour réaliser une affectation ne doit en aucun cas être confondu avec un symbole d'égalité tel qu'il est compris en mathématique. Il est tentant d'interpréter l'instruction **hauteur\_du\_lac = 375** comme une affirmation d'égalité, mais ce n'en est pas une !

- Premièrement, l'égalité est commutative, alors que l'affectation ne l'est pas. Ainsi, en mathématique, les écritures **a = 7** et **7 = a** sont équivalentes, alors qu'une instruction de programmation telle que **375 = hauteur\_du\_lac** serait illégale.
- Deuxièmement, l'égalité est permanente, alors que l'affectation peut être remplacée comme nous venons de le voir. Lorsqu'en mathématique, nous affirmons une égalité telle que **a = b** au début d'un raisonnement, alors **a** continue d'être égal à **b** durant tout le développement qui suit.

En programmation, une première instruction d'affectation peut rendre égales les valeurs de deux variables, et une instruction ultérieure en changer ensuite l'une ou l'autre. Exemple :

```
a = 2
b = a      # a et b contiennent des valeurs égales
b = 7      # a et b sont maintenant différentes
```

Rappelons ici, comme vous l'avez constaté dans les exercices du chapitre 1, que Python permet d'affecter des valeurs à plusieurs variables simultanément :

```
a, b, c = 2, 7, 4
```

## Répétitions en boucle – L'instruction while

En programmation, on appelle **boucle** un système d'instructions qui permet de répéter un certain nombre de fois (voire indéfiniment) toute une série d'opérations. Python propose deux instructions particulières pour construire des boucles : l'instruction **for ... in ...**, très puissante, que nous étudierons au chapitre 8, et l'instruction **while** que nous allons découvrir tout de suite.

Veuillez donc entrer les commandes ci-dessous :

```
passagers = 0
while (passagers < 7):
    passagers = passagers + 1
    print(passagers)
print("Demain 17 partira une très-bonne voiture pour Paris.")
```

Lancez l'exécution du bloc. Que se passe-t-il ? Avant de continuer plus loin, décrivez le résultat obtenu, et essayez de l'expliquer de la manière la plus détaillée possible.

L'instruction **while** (« tant que »), utilisée à la seconde ligne, indique à Python qu'il lui faut répéter *continuellement* le bloc d'instructions qui suit, *tant que* le contenu de la variable **passagers** reste inférieur à 7.

- Avec l'instruction **while**, Python commence par évaluer la validité de la *condition* fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
- Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.

Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant *le corps de la boucle*, c'est-à-dire :

- l'instruction **passagers = passagers + 1** qui incrémente d'une unité le contenu de la variable **passagers** (ce qui signifie que l'on affecte à la variable **passagers** une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
- l'appel de la fonction **print()** pour afficher la valeur courante de la variable **passagers**.

Lorsque ces deux instructions ont été exécutées, la première *itération* est terminée, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction **while**. La condition qui s'y trouve (ici **passagers < 7**) est à nouveau évaluée, et ainsi de suite.

Comme l'instruction **if** abordée au chapitre précédent, l'instruction **while** amorce une instruction composée. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Lorsque vous examinez un problème de cette nature, vous devez considérer les lignes d'instruction, bien entendu, mais surtout décortiquer *les états successifs des différentes variables* impliquées dans la boucle. Cela n'est pas toujours facile. Pour vous aider à y voir plus clair, vous pouvez prendre le temps de noter sur papier l'état des différentes variables.

Quelques remarques supplémentaires :

- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).
- Si la condition est fausse au départ, le corps de la boucle n'est *jamais* exécuté.
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment et l'exécution du bloc ne se termine jamais. Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par while, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.

Exemple de boucle sans fin (à éviter !) :

```
passagers = 0
while (passagers < 7):
    print(passagers)
print("Demain 17 partira une très-bonne voiture pour Paris.")
```

Dans vos programmes futurs, vous serez très souvent amené-e-s à mettre au point des boucles de répétition comme celle que nous analysons ici. Il s'agit d'une question essentielle, que vous devez apprendre à maîtriser parfaitement. Soyez sûrs que vous y arriverez progressivement, à force d'exercices.

### Bonnes pratiques et commentaires

Un script Python contiendra des séquences d'instructions identiques à celles que vous avez expérimentées jusqu'à présent. Puisque ces séquences sont destinées à être conservées et relues plus tard par vous-même ou par d'autres, il vous est très fortement recommandé d'expliciter vos scripts le mieux possible, en y incorporant des **commentaires**. Le meilleur emplacement pour cette description est le corps même du script (ainsi elle ne peut pas s'égarer).

*Un-e bon-ne programmeur-euse veille toujours à commenter ses scripts. En procédant ainsi, non seulement il ou elle facilite la compréhension de ses algorithmes pour d'autres personnes, mais clarifie aussi ses propres idées.*

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère #. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Il est important d'inclure des commentaires *au fur et à mesure de l'avancement de votre travail de programmation*. N'attendez pas que votre script soit terminé pour les ajouter «

après coup ». Vous vous rendrez progressivement compte qu'un-e programmeur-euse passe beaucoup de temps à relire son propre code (pour le modifier, y rechercher des erreurs, etc.). Cette relecture sera grandement facilitée si le code comporte suffisamment d'explications et de remarques.

### **Exercices**

*1 – Transition démographique* Recopiez les lignes ci-dessous et lancez l'exécution du bloc. Expliquez chaque ligne aussi précisément que possible. Que se passe-t-il si l'on supprime la 5<sup>e</sup> ligne ? Comment l'expliquez-vous ? Que se passerait-il si l'on supprimait la 6<sup>e</sup> ligne ?

```
population, natalite, mortalite = 10000, 0.04, 0.02
annee = 1800
while annee < 1900:
    population = population * (1+natalite-mortalite)
    population = int(population)
    annee = annee + 1
    print(annee, ':', population)
```

*2 – Rue de Bourg* Rédigez un algorithme qui affiche tous les numéros jusqu'à 40 et affiche « pair » lorsque le numéro se trouve du côté pair de la rue et « impair » sinon.  
*Indice : pensez à l'opérateur modulo %.*

*3 – Le Petit Chêne* Rédigez un algorithme qui affiche la suite de symboles suivante :

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

```
*****
```

*Indice : les chaînes de caractères peuvent être concaténées à l'aide de l'opérateur + (exemple : "Charles" + " " + "Kohler" équivaut à "Charles Kohler")*

---

## 4. PRINCIPAUX TYPES DE DONNEES

Dans les précédents chapitres, nous avons déjà manipulé des données de différents types, comme des nombres entiers ou réels, et des chaînes de caractères. Il est temps à présent d'examiner d'un peu plus près ces types de données, et également de vous en faire découvrir d'autres.

### Les données numériques

Dans les exercices réalisés jusqu'à présent, nous avons déjà utilisé des données numériques de deux types : les nombres entiers ordinaires (*int*) et les nombres réels à virgule flottante (*float*). En Python, les nombres à virgules flottantes peuvent également être donnés en notation scientifique. Exemple :

```
1e6      # 100000
1e+6     # 100000
1e-6     # 0.000001
```

Notez que si vous travaillez avec des *float* très grands ou très petits, les résultats seront automatiquement imprimés en notation scientifique par l'interpréteur.

Comme vous avez déjà pu le remarquer dans les exercices du chapitre 3, les fonctions **int()** et **float()** permettent au besoin de passer d'un type de données à l'autre.

### Exercices

1      Imprimez la valeur et le type de chacune des variables ci-dessous. Que constatez-vous ?

```
a = int(1.2)
b = int(2.7)
c = int(3.5)
d = float(a)
```

2      Une légende de l'Inde ancienne raconte que le jeu d'échecs a été inventé par un vieux sage, que son roi voulut remercier en lui affirmant qu'il lui accorderait n'importe quel cadeau en récompense. Le vieux sage demanda qu'on lui fournisse simplement un peu de riz pour ses vieux jours, et plus précisément un nombre de grains de riz suffisant pour que l'on puisse en déposer 1 seul sur la première case du jeu qu'il venait d'inventer, deux sur la suivante, quatre sur la troisième, et ainsi de suite jusqu'à la 64<sup>e</sup> case.

Écrivez un programme Python qui affiche le nombre de grains à déposer sur chacune des 64 cases du jeu. Calculez ce nombre de deux manières : 1) en utilisant des nombres entiers, et 2) en utilisant des nombres réels à virgule flottante. Notez les différences d'affichage.

## Les données alphanumériques

Outre des nombres, un programme peut également traiter des caractères alphabétiques, des mots, des phrases, ou des suites de symboles quelconques. Dans la plupart des langages de programmation, il existe pour cet usage des structures de données particulières que l'on appelle « chaînes de caractères » (string).

Une donnée de type *string* peut se définir en première approximation comme une suite quelconque de caractères. Dans un script Python, on peut délimiter une telle suite de caractères soit par des apostrophes droits (simple quotes), soit par des guillemets droits (double quotes). Exemple :

```
prenom = "Francine"  
nom    = 'Fritaix'  
surnom = 'dite "Fanchette"'  
print(prenom, nom, ', ', surnom)
```

Les 3 variables **prenom**, **nom**, **surnom** sont donc des variables de type *string*. Quel résultat s'imprime lors de l'exécution ?

Remarquez l'utilisation des apostrophes pour délimiter une chaîne qui contient des guillemets. Inversement, il est aussi judicieux d'utiliser des guillemets pour délimiter une chaîne qui contient des apostrophes. Remarquez encore une fois que la fonction **print()** insère un espace entre les éléments affichés.

Attention à ne pas confondre l'apostrophe droit ' avec les symboles analogues tels que ', ` ou '. Le même constat s'applique pour les guillemets. Seuls les guillemets droits peuvent être utilisés pour délimiter une chaîne de caractères. Pas d'inquiétude toutefois, car Jupyter notebook sélectionnera automatiquement les bons apostrophes ou guillemets lors de la rédaction. Ce genre d'erreur arrive généralement plutôt après un copier-coller.

### Exercice

3      Imprimez séparément les trois chaînes de caractères ci-dessous. Que constatez-vous ? Quel double caractère permet d'insérer un saut de ligne ? Que se passe-t-il si vous retirez les *backslashes* ? Quel mécanisme d'**échappement** des caractères spéciaux identifiez-vous ?

```
text1 = 'On a perdu un "chien d\'arrêt" âgé de 5 ans environ.'  
text2 = "Il répond au nom de \"Bravo\""  
text3 = "Taille moyenne, poitrail large, fouet mince, \n manteau  
fond blanc tacheté brun avec une large plaque au flanc. \n On promet  
cent francs de Suisse de récompense à celui qui le ramènera."  
text4 = """On a perdu un "chien d'arrêt" âgé de 5 ans environ."""
```

Le caractère spécial « \ » (*backslash*) permet quelques subtilités complémentaires :

- En premier lieu, il permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- À l'intérieur d'une chaîne de caractères, le *backslash* permet d'échapper un certain nombre de codes spéciaux (sauts de ligne, apostrophes, guillemets, etc.) pour les insérer dans le corps du texte.

Comme vous le remarquez, les *triples guillemets* """ ou *triples apostrophes* " " permettent d'insérer plus aisément des caractères spéciaux ou « exotiques » dans une chaîne, sans faire usage du *backslash*.

### Accès aux caractères individuels d'une chaîne

Les chaînes de caractères constituent un cas particulier d'un type de données plus général que l'on appelle des **données composites**. Une donnée composite est une entité qui rassemble dans une seule structure un ensemble d'entités plus simples : dans le cas d'une chaîne de caractères, par exemple, ces entités plus simples sont évidemment les caractères eux-mêmes. En fonction des circonstances, nous souhaiterons traiter la chaîne de caractères, tantôt comme un seul objet, tantôt comme une collection de caractères distincts. Un langage de programmation tel que Python doit donc être pourvu de mécanismes qui permettent d'accéder séparément à chacun des caractères d'une chaîne. Comme vous allez le voir, ce n'est pas très compliqué.

Python considère qu'une chaîne de caractères est un objet de la catégorie des **séquences**, lesquelles sont des **collections ordonnées** d'éléments. Cela signifie simplement que les caractères d'une chaîne sont toujours disposés dans un certain ordre. Par conséquent, chaque caractère de la chaîne peut être désigné par sa place dans la séquence, à l'aide d'un **index**.

Pour accéder à un caractère bien déterminé, on utilise le nom de la variable qui contient la chaîne et on lui accolé, entre deux crochets [ ], l'index numérique qui correspond à la position du caractère dans la chaîne. Prenez le temps de vérifier où se trouvent les crochets sur votre clavier. Si vous n'avez pas de touche dédiée, essayez alt+5/alt+6.

Attention cependant : les données informatiques sont presque toujours *numérotées à partir de zéro* (et non à partir de un). C'est le cas pour les caractères d'une chaîne. Exemple :

```
ch = "Lausanne"
print(ch[0], ch[2], ch[5], ch[7])
```

Résultat : L u n e

## Opérations élémentaires sur les chaînes

Comme vous avez déjà pu l'expérimenter au travers des exercices, il est possible de **concaténer**, c'est-à-dire d'assembler des chaînes de caractères à l'aide de l'opérateur **+**. Les deux chaînes de caractères se retrouvent alors collées l'une à la suite de l'autre sans espace interstitiel. Il vous faudra donc cas échéant ajouter ces espaces vous-mêmes. Exemple :

```
prenom = "Francine"  
nom = 'Fritauch'  
surnom = 'dite "Fanchette"'  
personnage = prenom + ' ' + nom, ', ', surnom
```

Résultat : 'Francine Fritauch, dite "Fanchette"'

## Quelques fonctions sur les chaînes de caractères

Nous allons tout de suite introduire quelques fonctions pratiques lors de l'utilisation des chaînes de caractères. Vous connaissez déjà deux fonctions Python : **print()** et **type()**.

La fonction **len()** permet de retourner la longueur de la chaîne de caractères, c'est-à-dire le nombre de caractères. Exemple :

```
prenom = "Francine"  
len(prenom)
```

Résultat : 8

La chaîne **prenom** comporte bien 8 caractères (numérotés de 0 à 7).

La fonction **str.split(a, b)** permet de couper une chaîne de caractères **a** à chaque fois qu'une deuxième chaîne de caractères **b** est rencontrée. Exemple :

```
enfants = "Louis|Marie|Jean-Samuel"  
str.split(enfants, '|')
```

Résultat : ['Louis', 'Marie', 'Jean-Samuel']

Vous remarquez que le résultat prend une forme inhabituelle. Il s'agit en effet d'une *liste*. Nous verrons ce dont il s'agit dans la prochaine section.

## Exercices

- 4 Ecrivez un script qui compte le nombre de barres horizontales | dans un texte.
- 5 Ecrivez un script qui recopie une chaîne de caractères dans une nouvelle variable en insérant un tiret entre chaque caractère. "Lausanne" deviendra "L-a-u-s-a-n-n-e".
- 6 À l'aide de la fonction **str.replace(a, b, c)**, découvrez comment inverser le résultat de l'exercice précédent. Indice : **a** doit être le résultat de l'exercice précédent.

7 Quel est le résultat obtenu par le script ci-dessous ? Corrigez-le pour obtenir **49**.

```
num1 = '18'  
num2 = '31'  
print(num1 + num2)
```

### Les listes (première approche)

Les chaînes que nous avons abordées à la rubrique précédente constituaient un premier exemple de données composites. Il en existe plusieurs autres, parmi lesquelles les *listes*, les *tuples* et les *dictionnaires*. Nous n'allons aborder ici que le premier de ces trois types, et ce de façon assez sommaire. Il s'agit en effet d'un sujet vaste, sur lequel nous devrons revenir.

En Python, on peut identifier une liste comme une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets. Exemple :

```
lieux = [ 'Rue Martheray', 'Derrière Bourg', 1006]
```

Dans cet exemple, la valeur de la variable **lieux** est une liste.

Comme on peut le constater dans le même exemple, les éléments individuels qui constituent une liste peuvent être de types différents. Dans cet exemple, en effet, les deux premiers éléments sont des chaînes de caractères, mais le quatrième élément est un entier. Par ailleurs, un élément d'une liste peut lui-même être une liste !

Remarquons aussi que, comme les chaînes de caractères, les listes sont des séquences, c'est-à-dire des collections ordonnées d'objets. Les divers éléments qui constituent une liste sont donc toujours disposés dans le même ordre, et l'on peut accéder à chacun d'entre eux individuellement si l'on connaît son index dans la liste. Comme c'était déjà le cas pour les caractères dans une chaîne, la numérotation de ces index commence à *partir de zéro*, et non à partir de un.

Exemples :

```
print(lieux[1])
```

Résultat : 'Derrière Bourg'

À la différence de ce qui se passe pour les chaînes, qui constituent un type de données *non-modifiables*, il est possible de changer les éléments individuels d'une liste, ou de les remplacer :

```
lieux[2] = lieu[2]-3  
lieux[0] = 'Place St. Laurent'  
print(lieux)
```

Résultat : ['Place St. Laurent', 'Derrière Bourg', 1003]

La **fonction intégrée** `len()`, que nous avons déjà rencontrée à propos des chaînes, s'applique aussi aux listes. Elle renvoie le nombre d'éléments présents dans la liste :

```
len(lieux)
```

Résultat : 3

Il est également possible d'ajouter un élément à une liste, mais pour ce faire, il faut considérer que la liste est un **objet**, dont on va utiliser l'une des **méthodes**. Nous reviendrons sur ces concepts un peu plus loin dans ces notes, mais nous pouvons dès à présent montrer « comment ça marche » dans le cas particulier d'une liste :

```
lieux.append('Chaucrau')  
print(lieux)
```

Résultat : ['Place St. Laurent', 'Derrière Bourg', 1003, 'Chaucrau']

Dans la première ligne de l'exemple ci-dessus, nous avons appliqué la méthode **a.append(b)** (« ajouter ») à l'objet **lieux**, avec l'**argument** 'Chaucrau'. On peut comprendre que la méthode **a.append(b)** est une sorte de **fonction** qui est en quelque sorte *attachée* ou intégrée aux objets de type « liste ». L'argument que l'on utilise avec cette fonction est bien entendu l'élément que l'on veut ajouter à la fin de la liste.

Nous verrons plus loin qu'il existe ainsi toute une série de ces méthodes, c'est-à-dire des fonctions intégrées, ou plutôt « encapsulées » dans les objets d'un certain type. Notons simplement que l'on applique une méthode à un objet en *reliant les deux à l'aide d'un point*. (D'abord le nom de la variable qui référence l'objet, puis le point, puis le nom de la méthode, cette dernière toujours accompagnée d'une paire de parenthèses.)

Nous avons en réalité déjà utilisé deux méthodes dans la section précédente : la méthode **str.replace(a, b, c)** et la méthode **str.split(a, b)** qui, comme vous avez pu le vérifier s'appliquent à un objet de type string. Comme vous pouvez le tester par vous-même, il est possible de les appliquer directement en utilisant la même syntaxe que pour la méthode **a.append(b)** : **a.replace(b, c)** et **a.split(b)**.

## Exercices

8 Soient les listes suivantes :

```
jours = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

```
mois = ['Janvier', 'Février', 'Mars', 'Avril', 'Mai', 'Juin', 'Juillet', 'Août', 'Septembre',  
'Octobre', 'Novembre', 'Décembre']
```

Écrivez un petit programme qui crée une nouvelle liste **t3**. Celle-ci devra contenir tous les éléments des deux listes en les alternant, de telle manière que chaque nom de mois soit suivi du nombre de jours correspondant :

```
['Janvier', 31, 'Février', 28, 'Mars', 31, ... ]
```

9 Écrivez un programme qui recherche le plus grand élément présent dans une liste donnée à l'aide d'une boucle **while**. Par exemple, si on l'appliquait à la liste [32, 5, 12, 8, 3, 75, 2, 15], ce programme devrait afficher : 75.

## 5. FONCTIONS PRÉDÉFINIES

*L'un des concepts les plus importants en programmation est celui de **fonction**. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable d'ordonner une liste de chiffres du plus petit au plus grand, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la réécrire à chaque fois.*

### Importer un module de fonctions de base

Vous avez déjà rencontré plusieurs fonctions intégrées au langage lui-même, comme la fonction **len()**, par exemple, qui permet de connaître la longueur d'une chaîne de caractères ou la fonction **print()**. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité : vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles. Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des programmes séparés que l'on appelle des **modules**. Un module est un fichier qui regroupe un ensemble de fonctions. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle, par anglicisme, des **librairies**.

Le module **re**, par exemple, contient des fonctions destinées à faciliter le travail sur les chaînes de caractères. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from re import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant *toutes* les fonctions (c'est la signification du symbole **\***) du module **re**. Cela vous permet d'utiliser directement dans le script de nouvelles fonctions, comme par exemple la fonction **findall(pattern, string)**, qui permet de trouver une expression régulière (**pattern**) spécifique dans une chaîne de caractères, comme dans l'exemple ci-dessous.

```
findall('[0-9]+', 'Avenue d'Ouchy 15, 1006 Lausanne')
```

Résultat : ['15', '1006']

Comme vous le remarquez, cette ligne de code permet de récupérer toute suite de nombres présente dans un texte. Nous reviendrons sur les expressions régulières, qui sont un outil très pratique, dans le chapitre 10.

Cet exemple permet de mettre en exergue quelques caractéristiques des fonctions :

1. Une fonction apparaît sous la forme d'un nom quelconque associé à des parenthèses : **findall(...)**

2. Dans les parenthèses, on transmet un ou plusieurs **arguments**. E.g. : **findall(pattern, string)**
3. Une fonction peut fournir une **valeur de retour**, ici un objet liste : **['15', '1006']**

Pour chaque module, un mode d'emploi est en général disponible. Cette **documentation** peut prendre différentes formes. Pour le module **re**, la documentation est par exemple présente à cette adresse : <https://docs.python.org/3/library/re.html>. Lorsque vous découvrez une nouvelle librairie ou un nouveau module, n'hésitez pas à rechercher la documentation relative sur internet. Cette documentation est également intégrée en partie à Python lorsque vous installez un nouveau module. Vous pouvez à tout moment retrouver les modalités d'utilisation d'une fonction en tapant un point d'interrogation, suivi du nom de votre fonction dans une cellule du notebook :

```
?findall
```

La commande ci-dessus affiche par exemple le résultat suivant :

```
Signature: findall(pattern, string, flags=0)
Docstring:
Return a list of all non-overlapping matches in the string.

If one or more capturing groups are present in the pattern, return
a list of groups; this will be a list of tuples if the pattern
has more than one group.

Empty matches are included in the result.
File:      ~/opt/anaconda3/lib/python3.8/re.py
Type:       function
```

### Installer et importer un nouveau module avec conda

Python est actuellement le langage de programmation le plus utilisé au monde. Il est très apprécié pour sa simplicité et le système de notebooks, notamment. Il l'est également grâce aux nombreux modules *open-source* qui en font l'un des langages les plus utilisés pour des algorithmes de pointe comme l'apprentissage automatique et le fameux *deep learning*. L'une des plateformes de partage d'algorithmes les plus connues est *Github*. Cependant, de nombreux modules publiés sont accessibles directement sur votre ordinateur grâce à *conda*, qui permet une installation entièrement automatique.

Pour installer un nouveau module, ouvrez un **terminal** ou **invite de commande**. Vous en avez une à disposition sur l'application Anaconda Navigator : *QT Console*. Sur Mac, vous pouvez aussi utiliser l'application *Terminal*. Ouvrez l'invite de commande et tapez :

```
conda install python-levenshtein
```

Appuyez sur *Enter*, puis, lorsqu'un message s'affiche, acceptez l'installation en tapant *y* et *Enter*.

Nous venons d'installer un nouveau module nommé Levenshtein. Ce module permet simplement de calculer la **distance de Levenshtein** entre deux chaînes de caractères. La distance de Levenshtein permet de mesurer la proximité de deux mots ou segments de

texte, en comptant le nombre d'opérations de base (remplacement d'une lettre, ajout, déletion) pour passer d'un segment de texte à l'autre. Exemple :

```
import Levenshtein
print(Levenshtein.distance("Louis", "Louise"), end=", ")
print(Levenshtein.distance("Marterey", "Martheray"))
```

Résultat : 1, 2

### **Exercices**

1 Soit la liste suivante :

```
rues = ['debourg', 'ede boung', 'r martheray', 'rue de bour', 'marterey', 'marlereg']
```

Écrivez un petit programme qui sépare la liste de rues en deux listes différentes, en s'appuyant sur leur distance de Levenshtein. À l'intérieur de chaque liste, la distance de Levenshtein doit être plus petite ou égale à 6.

2 Appliquez la fonction **Levenshtein.median(list)** sur les deux sous-listes de l'exercice précédent et affichez le résultat.

## 6. FONCTIONS ORIGINALES

*La programmation est l'art d'apprendre à un ordinateur à accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles fonctions au langage de programmation que vous utilisez.*

### Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les principes du langage. Lorsque vous commencerez à développer vos projets, vous serez confrontés à des problèmes parfois complexes, et les lignes de programme commenceront à s'accumuler.

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Le concept de fonction a été inventé afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par apprendre à **définir une fonction** en Python.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nom_de_la_fonction(liste de paramètres) :  
    ...  
    Bloc d'instructions  
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « \_ » est permis).
- Comme les instructions **if** et **while** que vous connaissez déjà, l'instruction **def** est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- La liste de paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction (les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments).
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un **appel de fonction** est constitué du nom de la fonction suivi de parenthèses, comme vous le savez déjà. Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans

la définition de la fonction, bien qu'il soit aussi possible de définir pour ces paramètres des valeurs par défaut.

### Fonctions simples sans paramètres

```
def table_de_7():
    n = 1
    while (n < 13):
        print(n*7, end=' ')
        n = n+1
```

Avec ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 12 premiers termes de la table de multiplication par 7. Notez bien les parenthèses, le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom :

```
table_de_7()
```

Résultat : 7, 14, 28, 35, 42, 49, 56, 63, 70, 77, 84

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction. Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, nous pouvons déjà noter deux propriétés intéressantes des fonctions :

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en faisant appel un algorithme complexe à l'aide d'une commande unique, à laquelle vous pouvez donner un nom très explicite si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. De manière générale, une règle d'or de la programmation consiste à éviter tout *duplicata* de code. Si vous devez faire appel plusieurs fois au même algorithme dans un même projet, placez ce dernier dans une fonction. De cette manière, vous éviterez de devoir traquer chaque occurrence individuelle de l'algorithme en question lorsque vous devrez le modifier.

## Fonctions paramétriques

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table de multiplication par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un **argument**. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un **paramètre**. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction :

```
def table_de_x(x) :  
    n = 1  
    while n < 13 :  
        print(n*x, end=' ')  
        n = n+1  
table_de_x(9)
```

Résultat : 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108

## Utilisation d'une variable comme argument

Dans les 2 exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction **table\_de\_x()** est une constante (la valeur 9). Cela n'est nullement obligatoire. L'argument que nous utilisons dans l'appel d'une fonction peut être une variable lui aussi, comme dans l'exemple ci-dessous :

```
a = 0  
while a < 2 :  
    table_de_x(a)  
    a = a + 1
```

Notez bien que le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction. Ces noms peuvent être identiques si vous le voulez, mais vous devez comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent éventuellement contenir une valeur identique). Dans l'exemple ci-dessus, l'argument que nous passons à la fonction **table\_de\_x()** n'est que *le contenu* de la variable **a**.

## **Exercice**

1 La fonction **table\_de\_x(x)** est intéressante, mais elle n'affiche toujours que les douze premiers termes de la table de multiplication. Améliorez-la en ajoutant deux autres arguments de cette manière : **table\_de\_x(x, debut, fin)**, de sorte que **debut** indique le premier terme de la table à afficher et **fin** le dernier.

Remarques :

- Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.
- Pour faire court, lors de l'appel de la fonction, les arguments utilisés doivent être fournis *dans le même ordre* que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.

## **Variables locales, variables globales**

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des **variables locales** à la fonction. C'est par exemple le cas des variables **x**, **debut**, **fin**, **n** et de toutes les autres variables que vous avez déclarées à l'intérieur de la fonction dans l'exercice précédent.

Chaque fois que la fonction **table\_de\_x()** est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel **espace de noms**. Les contenus des variables sont stockés dans cet espace de noms qui est *inaccessible depuis l'extérieur de la fonction*. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable **debut** juste après avoir effectué l'exercice ci-dessus, nous obtiendrons un message d'erreur. L'espace de noms est en effet automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies à l'extérieur d'une fonction sont des **variables globales**. Leur contenu est « visible » à l'intérieur d'une fonction, mais il est fortement recommandé de ne pas essayer de le modifier. Préférez plutôt la création d'une nouvelle variable locale qui ne pourra pas interférer avec le reste de votre programme.

## **Valeur de retour**

Une fonction renvoie souvent une **valeur de retour**. Cela permet de conserver le(s) résultat(s) principal(aux) après la suppression de l'espace de noms de la fonction. L'instruction **return** définit ce qui doit être renvoyé par la fonction :

```

def table_de_x(x) :
    n = 1
    resultat = []
    while n < 13 :
        resultat.append(n*x)
        n = n+1
    return resultat
table = table_de_x(9)
table

```

Résultat : [9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108]

Notez qu'il est également possible pour une fonction de retourner plus d'une valeur. On pourrait par exemple imaginer une fonction qui ressemble à ceci :

```

def table_de_x_et_y(x, y) :
    ...
    return resultat_1, resultat_2
table_x, table_y = table_de_x_et_y(9, 3)

```

### Utilisation des fonctions dans un script

Nous savons désormais créer une nouvelle fonction et l'utiliser. Notons encore que les fonctions doivent évidemment être définies avant d'y faire appel. Ceci est également valable pour les fonctions importées d'un module. De manière générale, la structure des programmes Python se présente ainsi :

```

# Imports
import pandas
import cv2

# Définition des fonctions
def maFonction(param1):
    """ Une fonction qui permet de ... """
    ...
    return resultat

# Corps principal du code
var1, var2 = 1831, 'Kohler'
var3 = maFonction(var1)

```

Les imports se trouvent tous *au début*, dans la toute première cellule du notebook. Cela facilite la visualisation des **dépendances**. Ainsi, vous pourrez vérifier sans perdre de temps les librairies à installer pour utiliser un programme. Les fonctions sont ensuite généralement stockées dans la *deuxième cellule*, ce qui permet de désengorger le corps principal du code et de faciliter sa lecture. Notez aussi qu'une bonne pratique consiste à

débuter la fonction par une ligne de texte, délimitée par des triples guillemets, qui documente l'utilité principale de la fonction. Cette ligne est considérée comme un simple commentaire et pas comme une variable. Elle permet simplement de clarifier le contenu, et potentiellement les modalités d'utilisation de la fonction.

### Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument *par défaut* pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus. Exemple :

```
def printOwner(nom, vedette, statut=' '):
    print(statut + ((len(statut)>0)*' ') + vedette + ' ' + nom,
          end=', ')
printOwner('Isabelle Hignou', 'Mme')
printOwner('Isaac Hignou', 'M.', 'feu')
```

Résultat : Mme Isabelle Hignou, feu M. Isaac Hignou,

Lorsque l'on appelle cette fonction en ne lui fournissant que les deux premiers arguments, le troisième reçoit tout de même une valeur par défaut – ici une chaîne de caractères vide. Si l'on fournit les trois arguments, la valeur par défaut pour le troisième est tout simplement ignorée. Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans tous les cas, cependant, les paramètres avec une valeur par défaut doivent *toujours se trouver en fin de liste*. Dans le cas contraire, vous obtiendrez une erreur.

### Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis *exactement dans le même ordre* que celui des paramètres qui leur correspondent dans la définition de la fonction. Python autorise cependant une souplesse beaucoup plus grande. On peut en effet faire appel à la fonction en fournissant les arguments correspondants *dans le désordre*, à condition de désigner nommément les paramètres correspondants. Exemple :

```
printOwner(vedette='M.', statut='fils de feu', nom='Isaac Hignou')
printOwner('Isabelle Hignou', statut='Hoirie de', vedette='Mme')
```

Résultat : fils de feu M. Isaac Hignou, Hoirie de Mme Isabelle Hignou,

## Exercices

1 – Géocodage À l'aide du terminal, installez le module **geopy**. Inspirez-vous ensuite de l'exemple ci-dessous pour programmer une fonction nommée **georefStreet** qui prend en argument un nom de rue et un nom de ville. Le nom de ville doit prendre la valeur par défaut *Lausanne*. Votre fonction doit retourner deux valeurs : **latitude** et **longitude**.

```
from geopy.geocoders import Nominatim  
geolocator = Nominatim(user_agent="cours_LTM")  
location = geolocator.geocode("175 5th Avenue, New York City")
```

2 – Géocodage Utilisez la fonction de l'exercice précédent pour créer deux listes, nommées **street\_latitude** et **street\_longitude**, en itérant sur la liste **street = ['Bourg', 'St Laurent', 'Cité devant', 'Moulins de Pépinet', 'Derrière Bourg']**. Utilisez l'instruction ci-dessous pour gérer les erreurs de géocodage :

```
if location is None:
```

3 – Géocodage Adaptez la fonction **georefStreet** pour qu'elle fournisse une troisième valeur de retour nommée **adresse**. Créez une deuxième fonction **checkLocation** qui vérifie que l'adresse trouvée par le géocodateur *Nominatim* se trouve effectivement à Lausanne. Vous pouvez tester votre fonction avec la liste **street = ['Bourg', 'St Laurent', 'Grand St Jean']**.

## 7. MANIPULER DES FICHIERS

Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'informations plus importante.

### Travailler avec des fichiers

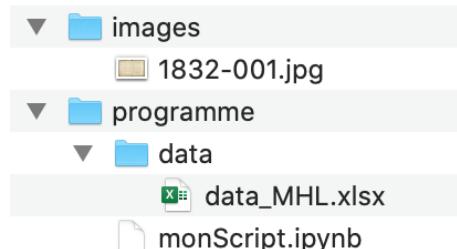
L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver (à l'aide de son titre), puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses, et vous pouvez aussi y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de page. Vous lisez la plupart des livres en suivant l'ordre normal des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre. Comme vous allez bientôt le découvrir, tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques.

### Chemin relatif

Le concept de chemin est essentiel pour trouver des fichiers sur votre ordinateur. Dans ce cours, nous n'allons aborder que le concept de *chemin relatif*. Le *chemin relatif* se base sur votre position actuelle dans l'ordinateur, c'est-à-dire la localisation du fichier Jupyter notebook en cours d'exécution. Prenons l'exemple du répertoire ci-dessous :

Pour aller du fichier **data\_MHL.xlsx** à partir du notebook **monScript.ipynb**, il faut d'abord entrer dans le répertoire **data** puis accéder au fichier **data\_MHL.xlsx**. Le chemin s'écrira comme cela :

```
'data/data_MHL.xlsx'
```



En revanche pour accéder à l'image **1832-001.jpg** depuis **monScript.ipynb**, il faut reculer d'un cran, accéder au répertoire **images** puis seulement faire appel à **1832-001.jpg**. Cela se traduit comme ceci :

```
'../images/1832-001.jpg'
```

### Lire les données shapefile

Les données **shapefile** (.shp) sont des données géographiques **vectorielles**. Vous manipulez des données vectorielles à chaque fois que vous utilisez un navigateur cartographique, comme *Google maps*. Ces données sont composées de *polygones*, de *lignes* et de *points*, à l'inverse des cartes stockées sous forme d'images (**raster**), qui sont composées de pixels. On peut lire les shapefiles avec des logiciels spécialisés, comme QGIS ou ArcGIS, mais il est parfois indispensable de passer par Python pour effectuer des

actions automatisées et spécifiques à un projet particulier. En tant que données vectorielles, les couches shapefile peuvent contenir des polygones, des lignes ou des points, mais jamais plus d'une de ces 3 catégories à la fois. Chacun de ces objets possède en général plusieurs attributs, sauvegardés dans une base de données (.dbf). La base de donnée DBF et son shapefile SHP sont indissociables l'une de l'autre et doivent donc toujours être stockées dans le même répertoire. Les attributs peuvent représenter des informations très différentes, comme un numéro de planche ou de parcelle cadastrale, le nom d'un propriétaire, ou encore une classe sémantique (e.g. bâtiment, champ, route, etc.).

Pour suivre l'exemple ci-dessous, vous aurez besoin d'installer pyshp :

```
conda install pyshp
```

Vous pourrez ensuite charger vos données dans Python comme ci-dessous. Vous remarquez que la fonction open() prend un argument un peu mystérieux nommé "rb+". Cela signifie « read binary (+ write) ». Pour éviter les erreurs, l'ouverture des fichiers sur Python est en effet conditionnée à une action *read* et/ou *write*. Lorsque vous ouvrez un fichier, vous devez donc savoir à l'avance si vous souhaitez lire ("r") le fichier, l'écrire ("w") ou les deux ("r+", "w+").

```
import shapefile  
shp = open("shapefile_1831/buildings_legende.shp", "rb+")  
dbf = open("shapefile_1831/buildings_legende.dbf", "rb+")  
sf = shapefile.Reader(shp=shp, dbf=dbf)  
print(sf)
```

```
Résultat : shapefile Reader  
    3307 shapes (type 'POLYGON')  
    3307 records (4 fields)
```

Nous avons donc chargé un ensemble shapefile/dbf qui contient 3307 shapes, toutes de type polygone, et chacune associée à 9 attributs. Il est possible de visualiser ces attributs :

```
sf.fields
```

```
Résultat : [('DeletionFlag', 'C', 1, 0), ['class', 'C', 12, 0],  
['layer', 'C', 254, 0], ['path', 'C', 254, 0], ['index', 'N', 50, 0],  
['page', 'N', 50, 0], ['number', 'N', 50, 0], ['use', 'C', 254, 0],  
['owner', 'C', 254, 0]]
```

On peut déchiffrer cette liste en imprimant un exemple :

```
sf.records()[2]
```

```
Résultat : Record #2: ['buildings', 'DecoupeBerney_001',
'/Users/remipetitpierre/Desktop/Cartes/Berney/vectorisation_2/shapefile/DecoupeBerney_001.shp', 4773, 1, 98, "['hangar']", 'Gaudin Schira']
```

Comme vous pouvez le constater, les champs sont divisés en différents types, soit chaîne de caractères (C), soit nombre entier (N). La troisième variable des listes **fields**, ci-dessus, correspond à la longueur maximale de la variable correspondante au champ.

On peut également afficher la géométrie des polygones :

```
sf.shape(2).points[:4]
```

```
Résultat : [(2537960.95, 1152430.25), (2537962.72, 1152435.80),
(2537973.27, 1152431.179), (2537972.81, 1152427.48)]
```

### Note sur les listes

Dans l'exemple ci-dessus, nous avons fait appel un nouvel opérateur sur lequel nous allons nous attarder brièvement. Nous avions vu qu'il était possible d'accéder à un élément d'une liste à l'aide de crochets :

```
sf.records()[2][7]
```

```
Résultat : 'Gaudin Schira'
```

Cependant, il est également possible et pratique de sélectionner un sous-ensemble d'éléments. Ceci est rendu possible par le double point (:).

```
liste[from:to]    # De from (compris) à to (non-compris)
liste[from:]      # De from (compris) à la fin
liste[:to]         # Du début à to (non-compris)
```

Il est important toutefois de se souvenir que la numérotation de la liste commence à toujours 0. De plus, l'indice situé à droite du double point (:) n'est pas compris dans la sélection.

```
sf.records()[2][4:7]
```

```
Résultat : [1, 98, "['hangar']"]
```

### Sauver des données shapefile

S'il est possible de charger des données shapefile sur Python et de les modifier, il est évidemment également possible de les sauvegarder. Cela se fait à l'aide d'un objet

Writer, dans la même perspective que nous avions utilisé un Reader pour lire les données dans un premier temps.

```
w = shapefile.Writer('full_lausanne_vector/Lausanne_v2')
w.fields = sf.fields[1:]
for shaperec in sf.iterShapeRecords():
    w.record(*shaperec.record)
    w.shape(shaperec.shape)
w.close()
```

Notez bien les différentes étapes de la procédure :

1. Créer un Writer vers un nouveau fichier (idéalement pas encore existant)
2. Annoncer la structure des données
3. Ajouter les géométries et les attributs au nouveau fichier
4. Fermer le Writer

Rappelez-vous de l'analogie du livre : elle s'applique à nouveau ici pour l'écriture, qui permet de sauvegarder vos opérations. Remarquez que votre shapefile est à nouveau sauvegardé en plusieurs fichiers (dont .shp et .dbf).

L'instruction **for ... in ...** est un nouveau type de boucle, que nous détaillerons dans le chapitre suivant.

### **Lire et générer des fichiers textes**

La lecture et l'écriture de fichiers texte fonctionne sur le même principe que les fichiers shapefile. Il s'agit en réalité du cas générique, qui permet non seulement d'agir sur des fichiers texte (.txt), mais également sur un très grand nombre de formats très différents, comme .shp, .xml, .json, .py, .html, etc. pour autant que vous connaissiez la syntaxe associée.

```
with open("test.txt", "w", encoding = 'utf-8') as file:
    file.write("1798 Rép. helvétique\n1803 Acte de Médiation\n")
    file.write("1813 Confédération des XXII cantons")
    file.close()
```

À nouveau, nous pouvons constater la présence de 3 phases : open, write, et close. Notez également l'instruction **with ... as ...**. Vous ne verrez probablement celle-ci que dans le contexte spécifique de lecture/écriture d'un fichier. L'utilisation de cette instruction est une bonne pratique, qui permet de s'assurer que le fichier soit refermé correctement à la fin de l'exécution du code. L'instruction **file.close()** est donc facultative dans cet exemple. La syntaxe **with ... as ...** permet de s'assurer que le fichier est bien refermé après l'écriture, même si une erreur survient durant l'exécution du code.

```
with open("test.txt", "r", encoding = 'utf-8') as file:  
    text = file.read()  
print(text)
```

Comme vous pouvez le constater dans l'exemple ci-dessus, la lecture du fichier est aussi simple que son écriture. Remarquez l'utilisation des paramètres "**w**" et "**r**" qui permettent, comme nous l'avons vu au début du chapitre, de spécifier le mode d'utilisation du fichier et donc de limiter les risques d'effacer celui-ci par erreur.

### **Exercices**

**1** Créez deux fonctions **storeList(...)** et **readList(...)**. La première doit permettre de stocker une liste donnée en argument dans un fichier .txt. La deuxième doit permettre d'ouvrir le fichier texte puis de retourner la liste d'origine.

**2** Créez une fonction qui prend en argument 1) un chemin vers un fichier shapefile, et 2) un nom de famille. La fonction doit permettre de lire le fichier, puis de créer un nouveau fichier shapefile, qui ne contient que les parcelles dont le nom de famille du ou de la propriétaire corresponde au nom donné en argument.

---

## 8. APPROFONDIR LES STRUCTURES DE DONNÉES

*Jusqu'à présent, nous avons travaillé sur des problèmes plus théoriques que pratiques, sur la base de petits exemples. Il est désormais temps de passer à l'échelle et d'apprendre à généraliser ces connaissances sur des bases de données entières. Il est également temps de vous faire découvrir des structures de données plus avancées.*

### Parcourir une séquence : les boucles **for**

Il arrive très souvent que l'on doive traiter l'intégralité d'une séquence (par exemple une liste ou une chaîne de caractères), pour effectuer sur chaque élément une opération quelconque. Nous appellerons cette opération un **parcours**. En nous limitant aux outils Python que nous connaissons déjà, nous pouvons envisager d'encoder un tel parcours à l'aide d'une boucle, articulée autour de l'instruction **while**. Cependant, comme le parcours d'une séquence est une opération très fréquente en programmation, il existe également une instruction plus directe pour réaliser cette opération : la boucle **for**.

```
metiers = ['journaliers', 'agriculteurs', 'ouvriers', 'artisans']
for metier in metiers:
    print(metier.capitalize())
```

Comme vous pouvez le constater, cette structure de boucle est plus compacte. Elle vous évite d'avoir à définir et à incrémenter une variable spécifique (un « compteur ») pour gérer l'indice du caractère que vous voulez traiter à chaque itération (c'est Python qui s'en charge). La structure **for ... in ...** ne montre que l'essentiel, à savoir que la variable **metier** contiendra successivement tous les items de la liste, du premier jusqu'au dernier.

Il est également possible de prédéfinir le nombre d'itérations de la boucle en utilisant la fonction **range(from, to, step)** qui permet de créer automatiquement une suite de nombres :

```
for year in range(1832, 1848, 1):
    print(year)
```

Comme vous pouvez le constater en reproduisant l'exemple ci-dessus, la valeur de **to** n'est pas comprise dans la suite de nombres. **range()** fonctionne donc sur le même principe que l'opérateur de sélection **[from:to]**.

### Appartenance d'un élément à une séquence : l'instruction **in** utilisée seule

L'instruction **in** peut être utilisée indépendamment de **for**, pour vérifier si un élément donné fait partie ou non d'une séquence. Vous pourriez par exemple vous servir de **in** pour vérifier si tel caractère alphabétique fait partie d'une liste ou d'une chaîne de caractères :

```
car = "e"
voyelles = "aeiouyàâéèëïîô"
if car in voyelles: # True
    ...

```

```
year = 1806
censuses = [1805, 1807, 1808, 1809, 1810, 1812, 1813]
if year in censuses: # False
    ...

```

## Les dictionnaires

Les types de données composites que nous avons abordés jusqu'à présent (*chaînes*, *listes*) étaient des séquences, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les **dictionnaires** que nous découvrons ici constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure, mais ce ne sont pas des séquences. Les éléments que nous allons y enregistrer ne seront pas disposés dans un ordre immuable. En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une **clé**, laquelle pourra être textuelle ou numérique.

Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaînes, des listes, des dictionnaires, et même des fonctions.

### Création d'un dictionnaire

À titre d'exemple, nous allons créer un dictionnaire des abréviations. Puisque le type dictionnaire est un type modifiable, nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un dictionnaire au fait que ses éléments sont enfermés dans une paire d'accolades. Un dictionnaire vide sera donc noté `{}` :

```
abbr = {}
abbr['jn'] = 'jean'
abbr['ff'] = 'fils de feu'
abbr['fs'] = 'françois'
print(abbr)
```

Résultat : `{'jn': 'jean', 'fs': 'françois', 'ff': 'fils de feu'}`

Comme vous pouvez l'observer dans le résultat, un dictionnaire apparaît dans la syntaxe Python sous la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades. Chacun de ces éléments est lui-même constitué d'une paire d'objets : une clé (index) et une **valeur**, séparées par un double point.

Veuillez à présent constater que l'ordre dans lequel les éléments apparaissent ou l'ordre dans lequel nous les avons fournis n'a aucune importance. Seule la clé permet d'accéder à une valeur du dictionnaire :

```
abbr[ 'jn' ]
```

Résultat : 'jean'

Remarquez aussi que contrairement à ce qui se passe avec les listes, il n'est pas nécessaire de faire appel à une méthode particulière (telle que **append()**) pour ajouter de nouveaux éléments à un dictionnaire : il suffit de créer une nouvelle paire clé-valeur.

La méthode **keys()** renvoie la séquence des clés utilisées dans le dictionnaire, qui s'apparente à une liste :

```
abbr.keys()
```

Résultat : dict\_keys(['ff', 'jn', 'fs'])

De manière analogue, la méthode **values()** renvoie la séquence des valeurs mémorisées dans le dictionnaire :

```
abbr.values()
```

Résultat : dict\_values(['fils de feu', 'jean', 'françois'])

Finalement, la méthode **items()** appliquée à un dictionnaire renvoie une séquence de tuples (clef, valeur). Le parcours effectué sur cette liste à l'aide de la boucle **for** permet d'examiner chacun de ces tuples un par un :

```
for ab, word in abbr.items():
    print(ab, word, end=', ')
```

Résultat : ff fils de feu, jn jean, fs françois,

## Pandas

Pour construire des projets plus importants, lorsqu'il vous faudra manipuler des bases de données structurées comportant de nombreuses variables, l'utilisation de listes imbriquées les unes dans les autres et des dictionnaires peut paraître peu efficace. En Python, il existe une librairie très populaire et intuitive, spécialisée dans la manipulation des bases de données : *Pandas*.

```
conda install pandas openpyxl
```

Cette librairie permet à la fois de lire et d'écrire les données très simplement, mais également de les structurer, de les analyser et de les transformer.

L'objet Pandas de base est un **DataFrame**, c'est à dire un tableau. Chaque DataFrame compte une ou plusieurs colonnes (columns) et lignes (rows). Un DataFrame qui ne compte qu'une seule colonne est appelé **Serie**. Il est possible de créer un nouveau DataFrame directement à partir d'un dictionnaire :

```
import pandas as pd
pd.DataFrame({'Nom': ['Pidoux', 'Golay', 'Dubochet'],
              'Naissance': [1902, 1888, 1923],
              'Terrain': [235.2, 1053.5, 456.0]})
```

|   | Nom      | Naissance | Terrain |
|---|----------|-----------|---------|
| 0 | Pidoux   | 1902      | 235.2   |
| 1 | Golay    | 1888      | 1053.5  |
| 2 | Dubochet | 1923      | 456.0   |

Les deux lignes ci-dessus suffisent à importer la librairie **pandas** (raccourcie **pd**), à créer un nouveau DataFrame et à l'afficher sur Jupyter. Comme vous pouvez le constater, le format d'affichage de pandas est moins aride que d'habitude, mais les avantages de cette librairie ne sont pas que cosmétiques. Elle facilite grandement la lecture et la sauvegarde des données ainsi que leur manipulation. Par exemple, pour lire et afficher les 3 premières lignes d'un fichier excel, deux lignes suffisent :

```
df = pd.read_excel('Legende_Berney/legende.xlsx', index_col=0)
df.head(n=3)
```

| folio | nr | articles | use | owner_surname             |
|-------|----|----------|-----|---------------------------|
| 0     | 1  | 1.0      | NaN | ['place', 'passage']      |
| 1     | 1  | 2.0      | NaN | ['terrasse']              |
| 2     | 1  | 3.0      | NaN | ['place']                 |
|       |    |          |     | Marie-Antoine De la Forêt |

Vous remarquez la présence du terme *NaN* dans certaines entrées. Dans Pandas et en Python de manière générale, *NaN* se rapporte à une donnée manquante.

Il est possible de sélectionner un sous-ensemble du DataFrame à l'aide de l'opérateur **[from:to]**. Les noms des colonnes peuvent être utilisés de la même manière que les clés d'un dictionnaire.

```
df['owner_surname'][46:49]
```

```
Résultat : 46    Charles-Antoine De Lerber
           47          Commune de Lausanne
           48    Marie-Antoine De la Forêt
Name: owner_surname, dtype: object
```

Il est possible d'appliquer des opérateurs de comparaison sur les colonnes. Cela renverra un **masque de booléens** (True/False) qui peut être utilisé pour sélectionner un sous-ensemble du DataFrame.

```
df['owner_surname'] == 'Commune de Lausanne'
```

Résultat : 0 True  
1 True  
2 False  
...

```
df[df['owner_surname'] == 'Charles-Amédée Kohler']
```

|      | folio | nr    | articles | use                            | owner_surname         |
|------|-------|-------|----------|--------------------------------|-----------------------|
| 1894 | 9     | 163.0 | NaN      | ['passage']                    | Charles-Amédée Kohler |
| 1895 | 9     | 164.0 | NaN      | ['maison']                     | Charles-Amédée Kohler |
| 1908 | 9     | 176.0 | NaN      | ['hangar']                     | Charles-Amédée Kohler |
| 1909 | 9     | 177.0 | NaN      | ['cour']                       | Charles-Amédée Kohler |
| 1910 | 9     | 178.0 | NaN      | ['maison', 'ecurie', 'remise'] | Charles-Amédée Kohler |

Comme vous pouvez le constater dans le deuxième exemple, pandas accepte comme opérateur de sélection une Serie de booléens, ce qui peut être très pratique pour accéder rapidement aux entrées qui respectent une certaine propriété que vous aurez définie. Par exemple, imaginons que vous vouliez sélectionner toutes les parcelles contenant une forge :

```
def hasUse(use):
    if 'forge' in use:
        return True
    else:
        return False
df[df['use'].apply(hasUse)].head(3)
```

|     | folio | nr    | articles | use                          | owner_surname        |
|-----|-------|-------|----------|------------------------------|----------------------|
| 44  | 1     | 44.0  | NaN      | ['maison', 'forge']          | George-Daniel Noir   |
| 121 | 1     | 119.0 | NaN      | ['forge', 'ecurie', 'fenil'] | Samuel-Daniel Rochat |
| 553 | 3     | 263.0 | NaN      | ['maison', 'forge']          | Jacob-César Collioud |

Comme vous pouvez le constater, il est facile d'appliquer rapidement une fonction à chaque ligne d'une colonne, grâce à la méthode **apply()**. Ici, nous cherchons à appliquer la fonction **hasUse()**, qui vérifie si '**forge**' se trouve ou non dans chaque entrée de la colonne **use**. Le premier paramètre de la fonction à appliquer est obligatoire et il prend successivement la valeur de chaque entrée de la colonne **use**.

Si l'on veut introduire des arguments supplémentaires, il faut utiliser la syntaxe peu conventionnelle `args=(arg1, arg2, ..., ).`

```
def hasUse(use, searchFor):
    if searchFor in use:
        return True
    else:
        return False
df[df['use'].apply(hasUse, args=('forge',))].head(3)
```

La librairie pandas possède également de nombreuses fonctions pratiques. Nous vous conseillons de parcourir attentivement la fiche-résumé à la page suivante pour mieux vous rendre compte des possibilités de cette librairie. Les fonctions s'appliquent en général directement sur des Serie ou des DataFrame. Par exemple, la fonction `df[col].value_counts()` permet de calculer le top 5 des plus grands propriétaires lausannois, par occurrence :

```
df['owner_surname'].value_counts()[:5]
```

```
Résultat : Commune de Lausanne      108
Etat du Canton de Vaud      61
Jean-Jaques Mercier fils    31
Godefroy-Jean Polier       19
Samuel-Daniel Rochat       19
Name: owner_surname, dtype: int64
```

### Exercices

1 Ouvrez le fichier excel des recensements 1832. Créez ensuite une fonction `cleanYears`, qui prend en argument une chaîne de caractères, correspondant à une entrée de la colonne `chef_annee_naissance`. Nettoyez la chaîne des symboles inopportuns, puis vérifiez si la longueur de la chaîne de caractères est bien 4, et si oui, tentez de la convertir en entier (`int`). L'âge du chef de famille doit être compris entre 14 et 100 ans. Si l'une de ces conditions n'est pas remplie, la fonction `cleanYears` doit retourner `np.nan` (vous aurez besoin de la librairie numpy, voir ci-dessous).

```
import numpy as np
```

2 Appliquez la fonction `cleanYears` à la colonne `chef_annee_naissance`. Sélectionnez ensuite uniquement les entrées de `chef_annee_naissance` qui ne prennent pas la valeur `nan`. (*Indice: une fonction pandas dédiée permet de se débarrasser des données nulles*). Affichez un histogramme de l'âge des chefs de famille.

3 Répétez la même opération avec la colonne `epouse_annee_naissance`. Pour les deux colonnes, affichez la moyenne, la médiane et la répartition par quantile. Que constatez-vous ? Si nécessaire, modifiez votre code pour afficher les deux distributions dans une même figure, et ainsi mettre en évidence les résultats. Quelles hypothèses socio-démographiques peuvent expliquer ces données ?

# Data Wrangling

## with pandas

### Cheat Sheet

<http://pandas.pydata.org>

#### Syntax – Creating DataFrames

| a | b | c |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

```
df = pd.DataFrame(
    {'a' : [4 ,5, 6],
     'b' : [7, 8, 9],
     'c' : [10, 11, 12]},
    index = [1, 2, 3])
Specify values for each column.
```

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
Specify values for each row.
```

| n | a | b | c  |
|---|---|---|----|
| 1 | 4 | 7 | 10 |
| 2 | 5 | 8 | 11 |
| 3 | 6 | 9 | 12 |

```
df = pd.DataFrame(
    {"a" : [4 ,5, 6],
     "b" : [7, 8, 9],
     "c" : [10, 11, 12]},
    index = pd.MultiIndex.from_tuples(
        [('d',1),('d',2),('e',2)],
        names=['n', 'v']))
Create DataFrame with a MultiIndex
```

#### Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
      .rename(columns={'variable' : 'var',
                       'value' : 'val'})
      .query('val >= 200')
     )
```

#### Summarize Data

```
df['w'].value_counts()
Count number of rows with each unique value of variable
len(df)
```

# of rows in DataFrame.

```
df['w'].unique()
# of distinct values in a column.
```

```
df.describe()
```

Basic descriptive statistics for each column (or GroupBy)



pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

```
sum() Sum values of each object.
count() Count non-NA/null values of each object.
median() Median value of each object.
quantile([0.25, 0.75]) Quantiles of each object.
apply(function) Apply function to each object.
```

```
min() Minimum value in each object.
max() Maximum value in each object.
mean() Mean value of each object.
var() Variance of each object.
std() Standard deviation of each object.
```

#### Group Data

```
df.groupby(by="col")
Return a GroupBy object,
grouped by values in column
named "col".
```

```
df.groupby(level="ind")
Return a GroupBy object,
grouped by values in index
level named "ind".
```

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

```
size() Size of each group.
```

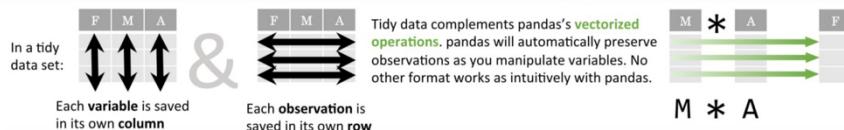
```
agg(function) Aggregate group using function.
```

#### Windows

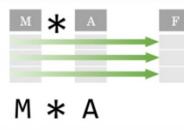
```
df.expanding()
Return an Expanding object allowing summary functions to be
applied cumulatively.
```

```
df.rolling(n)
Return a Rolling object allowing summary functions to be
applied to windows of length n.
```

#### Tidy Data – A foundation for wrangling in pandas

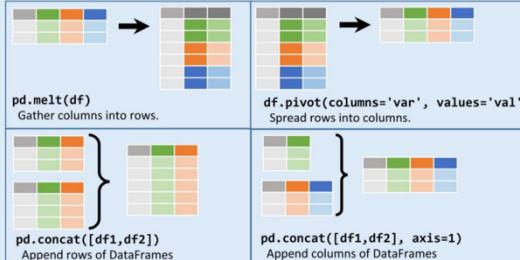


Tidy data complements pandas's **vectorized operations**. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



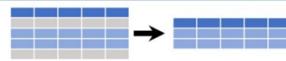
**M \* A**

#### Reshaping Data – Change the layout of a data set



```
df.sort_values('mpg')
Order rows by values of a column (low to high).
df.sort_values('mpg', ascending=False)
Order rows by values of a column (high to low).
df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame
df.sort_index()
Sort the index of a DataFrame
df.reset_index()
Reset index of DataFrame to row numbers, moving
index to columns.
df.drop(['Length','Height'], axis=1)
Drop columns from DataFrame
```

#### Subset Observations (Rows)



```
df[df.Length > 7]
Extract rows that meet logical
criteria.
df.duplicated()
Remove duplicate rows (only
considers columns).
df.head(n)
Select first n rows.
df.tail(n)
Select last n rows.
```

| Logic in Python (and pandas) |                        |                             |
|------------------------------|------------------------|-----------------------------|
| <                            | Less than              | !=                          |
| >                            | Greater than           | df.column.isin(values)      |
| ==                           | Equals                 | df.isnull(obj)              |
| <=                           | Less than or equals    | df.notnull(obj)             |
| >=                           | Greater than or equals | &,  , ~, df.any(), df.all() |

#### Subset Variables (Columns)



```
df[['width', 'length', 'species']]
Select multiple columns with specific names.
df['width'] = df.width
Select single column with specific name.
df.filter(regex='regex')
Select columns whose name matches regular expression regex.
```

| regex (Regular Expressions) Examples |   |
|--------------------------------------|---|
| '.'                                  | Matches strings containing a period '.'.                      |
| 'Length\$'                           | Matches strings ending with word 'Length'.                    |
| '^Sepal'                             | Matches strings beginning with the word 'Sepal'.              |
| '^x[1-5]\$'                          | Matches strings beginning with 'x' and ending with 1,2,3,4,5. |
| '[^Species].*''                      | Matches strings except the string 'Species'.                  |

```
df.loc[:, 'x2': 'x4']
Select all columns between x2 and x4 (inclusive).
df.iloc[:, [1, 2, 5]]
Select columns in positions 1, 2 and 5 (first column is 0).
df.loc[df['a'] > 10, ['a', 'c']]
Select rows meeting logical condition, and only the specific columns .
```

This cheat sheet inspired by RStudio Data Wrangling CheatSheet (<https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>) Written by Irv Lustig, Princeton Consultants

#### Handling Missing Data

```
df.dropna()
Drop rows with any column having NA/null data.
```

```
df.fillna(value)
Replace all NA/null data with value.
```

#### Make New Columns

```
df.assign(Area=lambda df: df.Length*df.Height)
Compute and append one or more new columns.
df['Volume'] = df.Length*df.Height*df.Depth
Add single column.
pd.qcut(df.col, n, labels=False)
Bin column into n buckets.
```

```
pd.agg(Vector function)
Vector function
```

pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

```
max(axis=1) min(axis=1)
Element-wise max. Element-wise min.
clip(lower=-10,upper=10) abs()
Trim values at input thresholds. Absolute value.
```

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

```
shift(1) shift(-1)
Copy with values shifted by 1. Copy with values lagged by 1.
rank(method='dense') cumsum()
Ranks with no gaps. Cumulative sum.
rank(method='min') cummax()
Ranks. Ties get min rank. Cumulative max.
rank(pct=True) cummin()
Ranks rescaled to interval [0, 1]. Cumulative min.
rank(method='first') cumprod()
Ranks. Ties go to first value. Cumulative product.
```

#### Combine Data Sets



**Standard Joins**

```
x1 x2 x3 pd.merge(adf, bdf,
               how='left', on='x1')
```

Join matching rows from bdf to adf.

```
x1 x2 x3 pd.merge(adf, bdf,
               how='right', on='x1')
```

Join matching rows from adf to bdf.

```
x1 x2 x3 pd.merge(adf, bdf,
               how='inner', on='x1')
```

Join data. Retain only rows in both sets.

```
x1 x2 x3 pd.merge(adf, bdf,
               how='outer', on='x1')
```

Join data. Retain all values, all rows.

**Filtering Joins**

```
x1 x2 adf[adf.x1.isin(bdf.x1)]
```

All rows in adf that have a match in bdf.

```
x1 x2 adf[~adf.x1.isin(bdf.x1)]
```

All rows in adf that do not have a match in bdf.

**Set-like Operations**

```
ydf zdf pd.merge(ydf, zdf)
Rows that appear in both ydf and zdf
(Intersection).
```

```
x1 x2 pd.merge(ydf, zdf, how='outer')
Rows that appear in either or both ydf and zdf
(Union).
```

```
x1 x2 pd.merge(ydf, zdf, how='outer',
                indicator=True)
.query('_merge == "left_only")
```

```
.drop(['_merge'], axis=1)
```

Rows that appear in ydf but not zdf (Setdiff).

#### Plotting

```
df.plot.hist()
Histogram for each column
```

```
df.plot.scatter(x='w',y='h')
Scatter chart using pairs of points
```



This cheat sheet inspired by RStudio Data Wrangling CheatSheet (<https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>) Written by Irv Lustig, Princeton Consultants

## 9. CARTOGRAPHIER ET VISUALISER

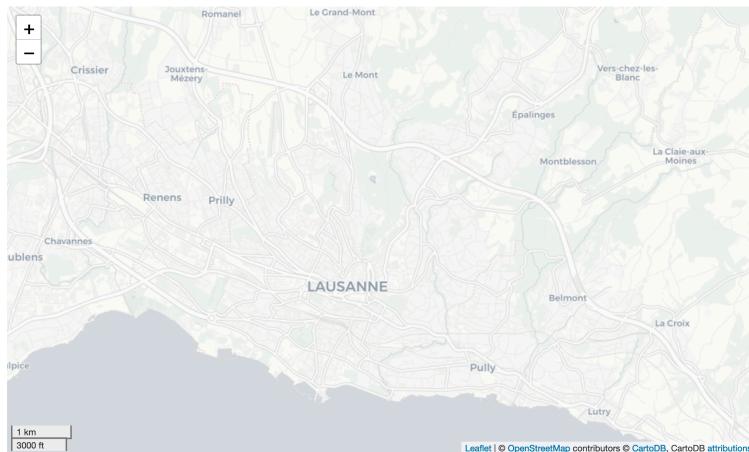
*La représentation des données est un enjeu crucial de l'histoire urbaine digitale et, plus généralement, des humanités digitales. Les données historiques s'incarnent sur une toile temporelle, mais aussi spatiale. La carte interactive est la première interface d'accès aux données. Elle permet de créer un lien entre les big data du passé et notre époque.*

### Leaflet

L'une des librairies les plus simples et complètes pour créer des cartes interactives est *folium*. Cette librairie s'appuie sur un module Javascript appelé *Leaflet*, qui permet de créer des cartes dynamiques. *folium* permet de visualiser des cartes interactives *Leaflet* directement dans un Jupyter notebook, ou de les exporter pour les intégrer par exemple à un site web. Toutefois, comme vous pourrez le remarquer, *folium* s'appuie sur des géodonnées contemporaines, notamment issues du "Wikipedia cartographique" : OpenStreetMap. La dimension temporelle est, encore aujourd'hui, totalement absente des grands logiciels GIS, en raison de la rareté des données historiques numérisées de qualité. Nous allons donc devoir composer en superposant plusieurs couches de données, ce qui n'est pas toujours facile, comme vous pourrez le constater en raison de la difficulté à géoréférencer correctement les données historiques.

```
conda install folium

import folium
lausanne_map = folium.Map(location=[46.53, 6.64], zoom_start=13,
                           tiles='CartoDB positron', control_scale=True)
lausanne_map
```



L'instruction ci-dessus vous permet de générer très facilement une carte interactive de Lausanne. La position du centre de la carte est encodée en coordonnées WGS84, qui est le *système de coordonnées* mondial utilisé pour le web. Nous reviendrons sur ce point plus tard. Vous remarquerez également qu'il est nécessaire de préciser un niveau de **zoom**. Le zoom est un paramètre très important des cartes numériques. En effet, les cartes numériques sont stockées sous la forme de tuiles. Une **tuile** peut être appréhendée

intuitivement comme un carreau de mosaïque dans lequel une zone de la carte est stockée à une certaine **échelle**. Lorsque votre navigateur charge la carte, seules les tuiles de la zone et de l'échelle correspondante sont chargées. Sans tuilage, il serait tout simplement impossible de charger une carte sur votre navigateur, en raison de la masse immense de données.

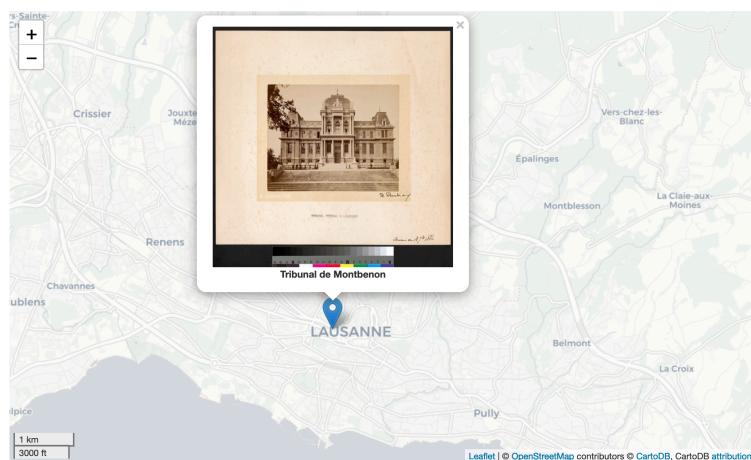
Outre la possibilité de manipuler des cartes interactives déjà existantes, folium permet de personnaliser ces dernières en y ajoutant nos propres données. L'élément le plus simple est le **marqueur**. Exemple :

```
folium.Marker([46.5209989, 6.6320879], popup = 'Charles  
Kohler').add_to(lausanne_map)
```

## Contenu HTML

Pour ajouter du contenu plus avancé, comme par exemple une image ou une vidéo, vous pouvez utiliser la syntaxe HTML (web) :

```
url = "http://museris.lausanne.ch/SGCM/GetImage.ashx?fileName=\  
SGCMImages/ImgHD/ImgDeg/C_MHL134056_1.JPG&id=134056"  
h, w = str(300), str(300)  
text = 'Tribunal de Montbenon'  
  
folium.Marker([46.5219771, 6.628651], popup = f'<br><center><b>{text}</b></center>'  
).add_to(lausanne_map)
```



Dans cet exemple, nous créons un objet web de type image, grâce à la syntaxe HTML. L'image est directement chargée à partir de son **url**. L'url est la source de l'image (src). Elle est insérée grâce à la **balise** `<img .../>`. Ici, on a également ajouté un texte en tant que légende. Les balises `<b>` et `<center>` sont simplement des indications de **formatage** qui permettent de centrer et d'afficher le texte en gras.

De manière très similaire, et pour rendre la carte encore plus interactive, vous pouvez facilement inclure une vidéo dans un marqueur :

```

url = "https://go.epfl.ch/cineac_video.mp4"
h, w = str(240), str(320)

folium.Marker([46.5233874, 6.6337036], popup = f'<video width="{w}"'
height="{h}" controls><source src="{url}" type="video/mp4">\'
</video>').add_to(lausanne_map)

```

## Clusters



Les points d'intérêt d'une ville se concentrent souvent dans les mêmes lieux. Par conséquent, vous pouvez facilement vous retrouver avec des marqueurs superposés. Pour éviter ce phénomène, qui vous empêcherait de cliquer sur certains marqueurs, il est recommandé et très souvent nécessaire de recourir à une visualisation par cluster.

```

from folium.plugins import MarkerCluster
marker_cluster = MarkerCluster().add_to(lausanne_map)

for i in range(5):
    folium.Marker(location=[46.5197, 6.6296], popup=f"marqueur \
{str(i)}").add_to(marker_cluster)

```

## Superposer une image

Il est possible de projeter l'image d'une carte historique directement sur la carte actuelle. Toutefois, toutes les cartes ne sont pas adaptées à cette manipulation car les relevés historiques souffrent souvent d'imprécision. La variabilité de la carte historique peut aussi découler des processus de gravure, d'impression, ou simplement de la déformation du papier au fil du temps.

Pour superposer une carte historique, il est d'abord nécessaire de géoréférencer manuellement cette dernière, par exemple avec QGIS, ou avec un logiciel web comme georeferencer.com. Cette étape doit vous permettre de déterminer la *rotation* de l'image ainsi que ses *limites*.

Dans l'exemple ci-dessous, nous utilisons l'API de Gallica, le moteur de recherche de la Bibliothèque nationale de France (BnF). Une **API** est un *software* permettant l'échange direct de données entre un *fournisseur* de données et un *client*. Pour les images, la plupart des grandes institutions patrimoniales occidentales se sont mises d'accord sur un ensemble de conventions permettant d'unifier les formats et les procédures d'accès. Il s'agit de **IIIF**, *International Image Interoperability Framework*. Ces conventions permettent de manipuler et de requêter facilement n'importe quelle image numérisée précalculée de manière standardisée. L'API Image de Gallica fonctionne selon la syntaxe suivante :

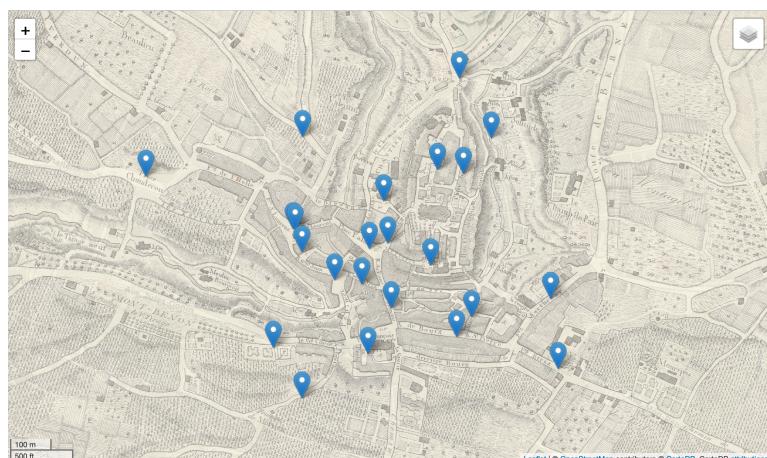
`{scheme}://{server}/{prefix}/{identifier}/{folio}/{image_region}/{size}/{rotation}/{quality}.{format}.`

<https://gallica.bnf.fr/iiif/ark:/12148/btv1b530292533/f1/641,804,5564,6941/full/22/native.jpg>

Dans cette requête url, nous demandons accès à l'image **b7v1b530292533** de la BnF, que l'on coupe dans les limites (641, 804), (5564, 6941). L'image est fournie en résolution maximale (full, native), avec une rotation de 22°.

Après avoir géoréférencé l'image avec QGIS, nous avons établi que la carte est délimitée par les coordonnées géographiques 46.5014 N 6.609 E / 46.538 N 6.6563 E.

```
folium.raster_layers.ImageOverlay(image='https://gallica.bnf.fr/iiif/ark:/12148/btv1b530292533/f1/641,804,5564,6941/full/22/native.jpg', bounds=[[46.5014, 6.609], [46.538, 6.6563]], opacity=0.5, interactive=False).add_to(lausanne_map)
```



## Le format GeoJSON

Nous avons déjà appris à manipuler les données géométriques shapefile dans le chapitre 7. Il existe un autre format couramment utilisé et qui vous permettra de dessiner des éléments vectoriels sur une carte Leaflet : le format **GeoJSON**. Comme shapefile, GeoJSON distingue trois types d'éléments vectoriels : les **points**, les **lignes** et le **polygônes**. Les données GeoJSON sont structurées comme un *dictionnaire*, ce qui permet de les manipuler facilement en Python.

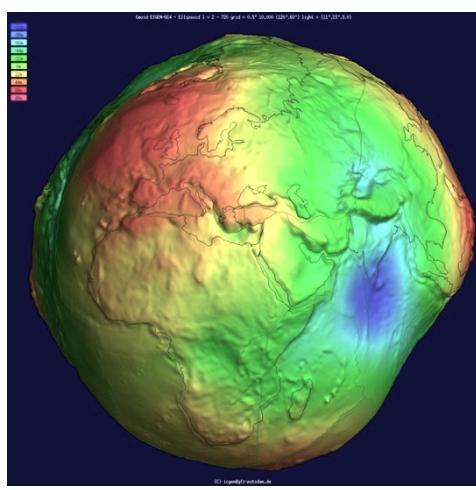
De plus, il est possible de passer très facilement d'un format shapefile à un format GeoJSON :

```
sf.shape(2).__geo_interface__
```

```
Résultat : {'type': 'Polygon',
'coordinates': [[[2537960.952523398, 1152430.2552146635),
(2537962.7244607382, 1152435.8021489454),
(2537973.2790440246, 1152431.1797037104), ...]
```

| Type       | Examples   |
|------------|--|
| Point      |  <pre>{   "type": "Point",   "coordinates": [30, 10] }</pre>  |
| LineString |  <pre>{   "type": "LineString",   "coordinates": [     [30, 10], [10, 30], [40, 40]   ] }</pre>   |
| Polygon    |  <pre>{   "type": "Polygon",   "coordinates": [     [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]   ] }</pre><br> <pre>{   "type": "Polygon",   "coordinates": [     [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],     [[20, 30], [35, 35], [30, 20], [20, 30]]   ] }</pre> |

## Coordinate Reference System (CRS)



Si elle n'est pas plate, la Terre n'est pas non plus réellement sphérique. Il s'agit plutôt d'une sorte d'ovoïde, relativement irrégulier. La représentation ci-contre, qui amplifie les déformations, permet de se rendre compte de l'approximation que constitue l'utilisation d'un **système de coordonnées** planétaire, comme WGS84 (Pseudo-mercator). En pratique, les géographes recourent plutôt à des modèle régionaux et nationaux, qui permettent de compenser localement les irrégularités de l'ovoïde terrestre. Pour la Suisse, il s'agit du système CH1903+, centré sur la position de l'Observatoire de Berne (1'200'000/2'600'000).

Pour utiliser des données provenant de cartes locales dans une interface web, il est donc nécessaire de changer de projection, et donc de système de coordonnées. Pour effectuer cette manipulation, vous aurez besoin d'installer GDAL :

```
conda install gdal
```

La définition des projections géométriques des différents systèmes de coordonnées sort du cadre de ce cours et nous vous donnons donc ces définitions telles quelles. Au niveau de la programmation, vous remarquerez qu'il n'y a pas de difficulté particulière à appliquer un changement de coordonnées.

```

CH1903p_wkt = """ PROJCS["CH1903+ / LV95",
    GEOGCS["CH1903+", 
        DATUM["CH1903+", 
            SPHEROID["Bessel 1841",6377397.155,299.1528128,
                AUTHORITY["EPSG","7004"]], 
            AUTHORITY["EPSG","6150"]], 
        PRIMEM["Greenwich",0,
            AUTHORITY["EPSG","8901"]], 
        UNIT["degree",0.0174532925199433,
            AUTHORITY["EPSG","9122"]], 
        AUTHORITY["EPSG","4150"]], 
    PROJECTION["Hotine_Obllique_Mercator_Azimuth_Center"], 
    PARAMETER["latitude_of_center",46.9524055555556], 
    PARAMETER["longitude_of_center",7.439583333333333], 
    PARAMETER["azimuth",90],PARAMETER["rectified_grid_angle",90],
    PARAMETER["scale_factor",1],PARAMETER["false_easting",2600000], 
    PARAMETER["false_northing",1200000],UNIT["metre",1,
        AUTHORITY["EPSG","9001"]], 
        AXIS["Easting",EAST], 
        AXIS["Northing",NORTH], 
        AUTHORITY["EPSG","2056"]]""

wgs84_wkt = """
    GEOGCS["WGS 84",
        DATUM["WGS_1984",
            SPHEROID["WGS 84",6378137,298.257223563,
                AUTHORITY["EPSG","7030"]], 
            AUTHORITY["EPSG","6326"]], 
        PRIMEM["Greenwich",0,
            AUTHORITY["EPSG","8901"]], 
        UNIT["degree",0.01745329251994328,
            AUTHORITY["EPSG","9122"]], 
        AUTHORITY["EPSG","4326"]]""
"""

from osgeo import osr, gdal
old_cs = osr.SpatialReference()
old_cs.ImportFromWkt(CH1903p_wkt)
new_cs = osr.SpatialReference()
new_cs.ImportFromWkt(wgs84_wkt)
transform = osr.CoordinateTransformation(old_cs, new_cs)
print(transform.TransformPoint(2537771, 1152835)

```

Résultat : (46.5239146, 6.6276394, 0.0)

### Exercices

- 1 Importez les données shapefile du cadastre Berney (déjà utilisées pour les exercices du chapitre 7). À l'aide de **folium.Polygon**, redessinez les bâtiments sur la carte de Lausanne. Attribuez une couleur différenciée aux polygones, en fonction du fait que le bâtiment est résidentiel ('maison'), non-résidentiel, ou mixte.
- 2 Chargez les métadonnées du corpus iconographiques du Musée Historique à l'aide de Pandas. Créez une sous-base de données qui ne contient que les entrées dont l'image est disponible en ligne et qui ont été créées aux alentours de 1831 (+/- 10 ans). Regroupez les adresses uniques dans une liste séparée puis géoréférez-les à l'aide de Nominatim (voir exercices du chapitre 6). Fusionnez les adresses géoréférencées avec votre corpus iconographique 1831. Créez une carte interactive pour présenter vos données.

---

# 10. FOUILLER ET ÉTUDIER LES TEXTES

*Le texte est l'une des données les plus complexes à manipuler et sémantiser numériquement. Le natural language processing (NLP) fait d'ailleurs partie des principaux défis scientifiques contemporains. Alors que la fouille quantitative des textes ouvre de nouveaux champs d'exploration des big data, ces thématiques révolutionnent également les humanités, avec l'ère du "distant reading" et de la linguistique computationnelle. Dans ce chapitre, nous n'aurons pas la prétention d'aller aussi loin, mais nous vous introduirons des outils et des méthodes de base pour le traitement des bases des données textuelles.*

## Expressions régulières

Nous avons reporté ce sujet plusieurs fois, et comme vous avez déjà probablement pu le découvrir dans les chapitres précédents, les **expressions régulières** sont des outils particulièrement puissants. Si leur syntaxe peut paraître complexe, le pouvoir d'extraction que leur maîtrise vous conférera vaut largement l'effort.

Les expressions régulières sont particulièrement utiles lorsqu'il s'agit de chercher une expression structurée précise dans un texte. Ce texte peut être un autre code (e.g. une page web Wikipedia, ou une entrée de la base de données Museris), ou une source historique, comme un annuaire du commerce, un article de journal, ou un dictionnaire historique.

Commençons par revenir sur l'exemple du chapitre 5 :

```
findall('[0-9]+', 'Avenue d'Ouchy 15, 1006 Lausanne')
```

Résultat : ['15', '1006']

Dans cet exemple, l'expression régulière **[0-9]+** permettait d'extraire les nombres d'une chaîne de caractères (pour ensuite les transformer en *int* par exemple). En réalité, cette expression signifie : "Sélectionner *un ou plusieurs* (+) caractères se suivant et faisant partie de l'*ensemble* ([...]) des caractères *compris entre 0 et 9 (0-9)*".

- Les crochets ([...]) permettent de définir un ensemble non-ordonné de caractères, *au choix*.
- Le tiret (-) permet de signifier que l'on cherche un caractère dont le numéro Unicode est compris entre celui du caractère situé juste avant le tiret (e.g. 0) et celui du caractère situé juste après le tiret (e.g. 9). Il est possible d'obtenir le numéro Unicode d'un caractère à l'aide de la fonction intégrée **ord()**. Ainsi, la commande **ord('0'), ord('9')** nous indique que le caractère '0' porte le numéro 48, tandis que '9' porte le numéro 57. À titre d'exemple, 'a' porte le numéro 97, 'à' le numéro 224 et '!' le numéro 33. Veillez donc à toujours vérifier quels caractères se trouvent dans votre intervalle.
- Le signe + permet d'indiquer que l'on cherche *une ou plusieurs* occurrences (par exemple un ou plusieurs caractères faisant partie d'un ensemble donné).

Vous pouvez aussi rechercher une séquence précise. Par exemple, vous pourriez vouloir étudier la présence des femmes nommées dans les journaux lausannois du XIXème siècle. Pour ce faire, vous pourriez commencer par chercher le mot "Madame" (avec ou sans majuscule) suivi d'un espace, puis d'un nom commençant par une lettre majuscule. La syntaxe de votre requête ressemblera alors à ceci :

```
text = "Ce matin, Monsieur Larpin fera miser la maison qu'il a  
acquise de Madame Dupin née Mercier."  
re.findall('([Mm]adame )([A-Z][a-zA-ÿ\-\-]+)', text)
```

Résultat : [ ('Madame ', 'Dupin') ]

Les parenthèses servent à créer un groupe de caractères. Ici le caractère **M / m**, puis la suite de caractères "**adame**", dans cet ordre précis. À l'intérieur d'une expression régulière, les caractères spéciaux doivent être échappés. Ici, par exemple, le tiret est échappé (\-), pour ne pas être confondu avec son autre fonction, que nous avons découvert dans l'exemple précédent. Notez que les caractères accentués sont distincts des caractères non-accentués et qu'ils doivent donc être inclus expressément (à-ÿ).

Mettons maintenant que vous travailliez à une échelle très différente, par exemple sur un corpus réunissant tous les journaux historiques francophones. Vous pourriez vouloir rendre cette instruction plus efficace en évitant qu'elle ne stocke systématiquement le mot "Madame" dans les résultats. En réalité, le système des expressions régulières permet de simplement localiser le segment qui vous intéresse, tout en exploitant son contexte à l'aide des "coups d'oeil" : *lookbehind* (?<=) et *lookahead* (?=). Exemple :

```
re.findall('(?<=[Mm]adame )([A-Z][\-\w]+)', text)
```

Résultat : ['Dupin']

Dans cet exemple, l'expression se trouvant dans la première parenthèse n'est que contextuelle. Dans la deuxième parenthèse, nous avons utilisé un mécanisme pratique, avec le raccourci \w+ qui permet de capturer n'importe quel caractère alphabétique. Prenez le temps d'étudier cette expression régulière pour bien comprendre son fonctionnement.

Il est aussi possible d'utiliser les expressions régulières pour nettoyer certaines données ou les remplacer par d'autres. La fonction **re.sub** permet de remplacer une expression par une nouvelle chaîne de caractères, un peu à l'image de la fonction **replace**, que vous connaissez déjà, mais avec la puissance des expressions régulières.

```
re.sub('([\^\-\w ]+)', '@', text)
```

Dans cet exemple, tous les caractères *non-alphabétiques* du texte sont remplacés par un arobase @. Le signe circonflexe ^ permet la négation de l'expression qui suit. Littéralement, l'expression remplace donc tous les caractères non-alphabétiques, qui ne sont ni des tirets, ni des espaces. Pour les supprimer, on pourrait remplacer '@' par une chaîne de caractères vide.

## Entités nommées

Les **entités nommées** sont des expressions textuelles caractéristiques et reconnaissables : par exemple des noms propres, des noms de lieux, des dates, des noms d'organisations, etc. La reconnaissance et l'extraction des entités nommées constitue un enjeu majeur du NLP. En histoire urbaine, ces technologies peuvent être très utiles pour créer des liens entre des données textuelles, des lieux et des personnes.

```
pip install spacy && python -m spacy download fr_core_news_md
```

```
import spacy
nlp_fr = spacy.load('fr_core_news_md')
nentities = nlp_fr('Grand concert vocal et instrumental dans la
salle du Casino, le mardi 5, par Mme Pollet, harpiste de Paris, MM.
Girard et Lagoanère;').ents
[(ne.label_, ne.text) for ne in nentities]
```

Résultat : [ ('LOC', 'Casino'), ('PER', 'Mme Pollet'), ('LOC',
'Paris'), ('PER', 'MM. Girard'), ('PER', 'Lagoanère') ]

Les modèles NLP comme celui de **spacy** sont en général pré-entraînés sur des corpus très grands, comme par exemple Wikipedia. La reconnaissance des entités nommées est évidemment spécifique à la langue. En conséquence, les modèles pour certaines langues, comme l'anglais ou le chinois, sont très performants, tandis que d'autres le sont nettement moins. Le français se trouve en général dans une tranche intermédiaire.

## Latent Dirichlet Allocation

La Latent Dirichlet Allocation (LDA, à ne pas confondre avec la *Linear Discriminant Analysis*) est une méthode de **topic modelling** (modélisation thématique), qui permet de distinguer les différents thèmes d'un texte ou d'une série de textes en s'appuyant sur la distribution statistique des **tokens** et en particulier sur la co-occurrence de certains groupes de tokens. Un token est en général un mot ou un groupe de mots (par exemple un tag, un hashtag ou une entité nommée). On peut le considérer comme une sorte de brique élémentaire du langage, que l'on cherche à sémantiser. Chaque token est une occurrence individuelle d'un **type**. Par exemple le token *concert*, dans l'exemple ci-dessus, est une instance du type "*concert*". Un type est en somme une classe, qui peut se traduire par la présence d'un ou plusieurs tokens, ou même d'aucun.



Pour utiliser LDA sur des données textuelles, il est nécessaire de rendre ces dernières numériquement intelligibles. La manière la plus simple de faire ceci est de subdiviser le texte en segments plus petits, que l'on appellera *documents*. Un *document* peut par exemple correspondre à une phrase, une réplique, ou à une entrée dans votre base de données. Chaque *document* contient un ou plusieurs tokens. L'ensemble des types (ici l'ensemble des tokens uniques existants) constitue le *vocabulaire*. Un numéro est attribué à chaque type du vocabulaire. Par exemple, *académie* correspondra au numéro 0, *Acadie* au numéro 1, *acétone* au numéro 2, et ainsi de suite. Grâce à ce système, chaque *document* peut être représenté comme une liste de 0 et de 1. La longueur du vecteur correspondra au nombre de mots compris dans le vocabulaire. Si le document contient un token correspondant au  $n^{\text{ième}}$  type du vocabulaire, la  $n^{\text{ième}}$  entrée de la liste prendra la valeur 1. Tous les autres types, absents du *document* en question, prendront la valeur 0.

Cette transformation peut vous paraître difficile à comprendre ou peu intuitive. Nous vous proposons de l'apprendre par la pratique, en essayant d'identifier des clusters thématiques dans la base de données iconographique du Musée historique Lausanne. Une liste de mots-clés, attribués par les archivistes à chaque image, constituera nos *documents*.

```
conda install scikit-learn
```

```
import pandas as pd
import numpy as np
from sklearn.decomposition import LatentDirichletAllocation

df = pd.read_excel('../Icono/data_MHL.xlsx', index_col=0)
key_cols = ['Mots-clés principaux : ', 'Mots-clés secondaires : ']
df = df.dropna(subset = key_cols)

documents, keywords = [], []
for i, row in df.iterrows():
    tokens = []
    for key in key_cols:
        tokens += row[key].split(', ')
    keywords += tokens
    documents.append(tokens)
```

Dans cet exemple, nous commençons par charger la base des métadonnées, puis nous ne conservons que les entrées pour lesquelles les deux colonnes de mots-clés ne sont pas nulles. Nous créons deux listes : **documents** et **keywords**, en itérant très simplement sur chaque ligne (row) du DataFrame. Nous stockons chaque mot-clé dans la liste continue **keywords** et dans la liste des **documents**, qui stocke séparément les mots-clés de chaque entrée dans une sous-liste<sup>1</sup>.

---

<sup>1</sup> Notez qu'il existe des manières plus efficientes de créer ces deux listes, notamment en utilisant la méthode `apply`. Si vous aimez les challenges purement computationnels, vous pouvez vous amuser à les trouver de votre côté. Pour cet exemple, nous avons privilégié la notation la plus lisible.

```

vocabulaire = pd.Series(keywords).value_counts()
vocabulaire = vocabulaire [vocabulaire >= 3].sort_index().keys()

```

Dans un deuxième temps, nous comptons toutes les occurrences uniques des mots-clés et ne conservons que ceux qui apparaissent au moins 3 fois. En dessous de ce seuil, l'utilisation d'un modèle statistique est peu pertinente. Cette liste des mots-clés uniques apparaissant au moins 3 fois constitue le **vocabulaire**.

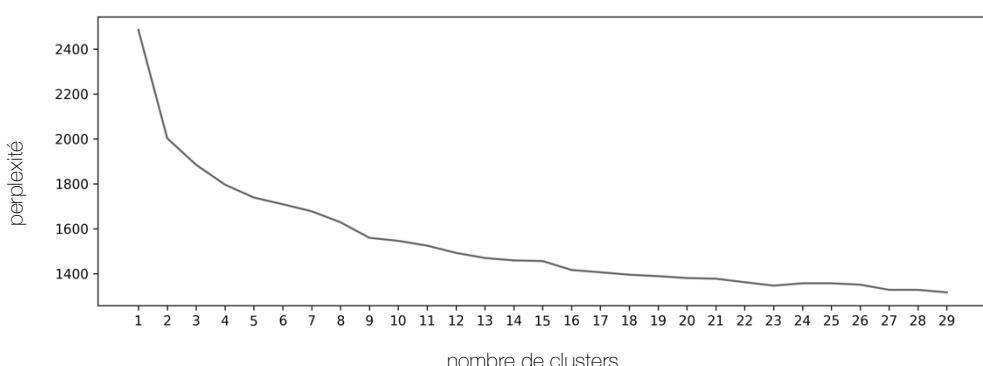
```

one_hot = np.zeros((len(documents), len(vocabulaire)))
for index, document in enumerate(documents):
    for token in document:
        one_hot[index][int(np.argmax(vocabulaire == token))] = 1

```

Pour chaque document, une liste de zéros de longueur **len(vocabulaire)** est initialisée. Une boucle itère ensuite sur chaque **token n** du **document** et remplace la **n<sup>ième</sup>** valeur de la liste par 1. Comme expliqué plus haut, n correspond au numéro attribué au type dudit token, dans le **vocabulaire**.

À présent, puisque nos données ont été traduites en langage numérique, il est possible d'appliquer la LDA. La LDA est un algorithme de clustering **paramétrique**. Il est en effet nécessaire de lui indiquer le nombre de **clusters** attendu. Dans le cas du *topic modelling*, chaque cluster correspond intuitivement à un thème. On pourrait donner à ce paramètre n'importe quelle valeur, mais il existe une métrique mathématique qui peut faciliter notre choix : la *perplexité*. La perplexité est une mesure de l'incohérence thématique. Sa validité est toutefois à considérer avec pragmatisme et précaution, car certaines études remettent en question sa cohérence avec la cognition humaine. Dans notre cas, la perplexité baisse logarithmiquement, de manière assez régulière. Nous pouvons donc simplement fixer le nombre de cluster qui nous convient en fonction du niveau de granularité que nous cherchons à atteindre.



Dans cet exemple, nous choisirons la valeur 9. Nous pouvons ensuite appliquer la LDA à nos données à l'aide de la fonction **fit\_transform()**.

```

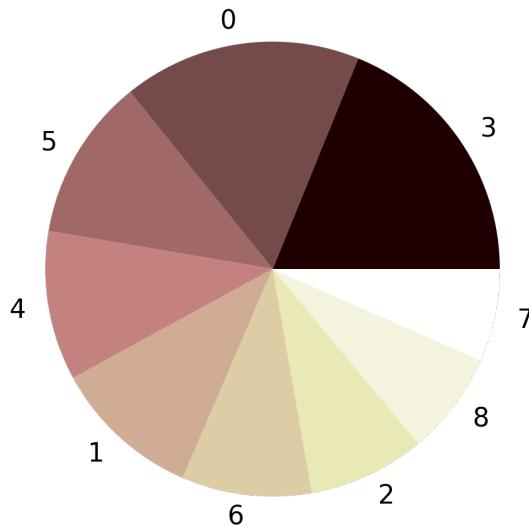
lda = LatentDirichletAllocation(n_components=9, random_state=0)
res = lda.fit_transform(one_hot)
print(lda.perplexity(one_hot))

```

Nous arrivons finalement à l'étape des résultats. Commençons par calculer rapidement l'attribution thématique de chaque image et la proportion des différents thèmes. Pour ceci, nous utiliserons la fonction **argmax** qui permet de récupérer l'indice de la valeur la plus élevée d'une liste. Dans notre cas, argmax permet de sélectionner *le thème le plus probable*, à partir d'une liste de probabilités. Nous utiliserons ensuite la fonctions de pandas **plot.pie()** pour représenter la distribution thématique à l'aide d'un camembert.

```
image_attribution = np.argmax(res, axis=1)
pd.Series(image_attribution).value_counts().plot.pie()
```

Résultat :



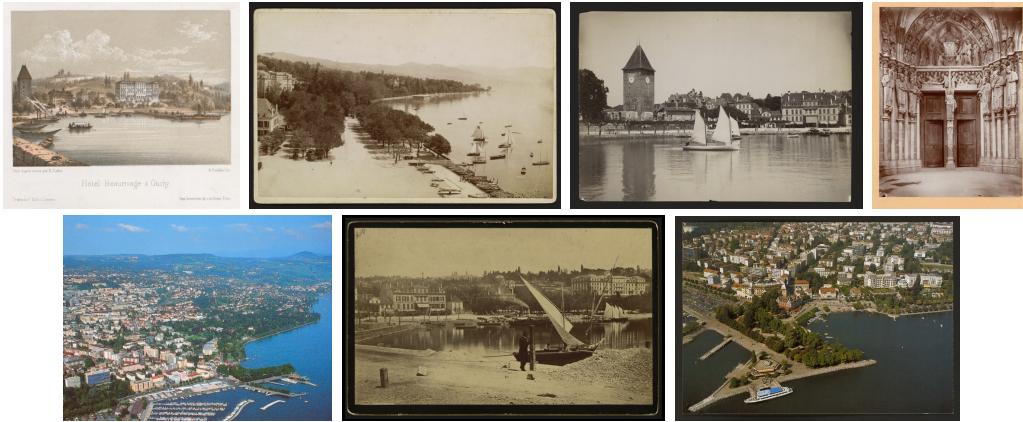
Evidemment, le travail herméneutique nous revient, les clusters sont simplement représentés par des numéros. Pour faciliter l'interprétation, nous pouvons par exemple visualiser les mots-clés les plus idiosyncratiques de chaque cluster. Exemple pour le cluster n°4 :

```
type_attribution = lda.transform(np.identity(len(vocabulaire)))
for t in vocabulaire[np.flip(np.argsort(type_attribution[
    :, 4]))[:10]]:
    print(t, end=' ', )
```

Résultat : Barque, Navigation, Voilier, Hôtel Beau-Rivage, Débarcadère, Tour d'Ouchy, Hôtel d'Angleterre, Allée des Bacounis, Quai de Belgique, CGN

On a donc affaire à un thème iconographique centré sur Ouchy, le lac, les quais et les Grands hôtels. On pourrait encore mieux comprendre ce thème en affichant les url des images les plus représentatives du cluster n°4 :

```
df['image_url'][np.flip(np.argsort(res[:, 4])[
    -10:])].dropna().apply(print)
```



Malgré quelques incohérences (e.g. la porte de la Cathédrale), la classification thématique paraît plutôt satisfaisante. Si vous le vouliez, vous pourriez répéter le processus ci-dessus plusieurs fois, en ajustant le nombre de classes, jusqu'à obtenir des clusters idéalement homogènes.

### **Exercices**

- 1 Importez la base de données cinématographique, puis créez deux nouvelles colonnes **lieux** et **personnes**. Pour chaque vidéo, récupérez les entités nommées se trouvant dans la description principale ou dans la description des séquences et ajoutez-les sous forme de liste dans la colonne **lieux** ou **personnes**.
- 2 Importez le fichier excel contenant la légende du cadastre Berney, déjà utilisé dans le chapitre 8. À l'aide des expressions régulières, récupérez les noms, prénoms et noms de naissance de toutes les femmes propriétaires mariées ou veuves.
- 3 Toujours à l'aide des expressions régulières, récupérez le nom du légataire et celui des hoirs pour toutes les propriétés appartenant à une hoirie.